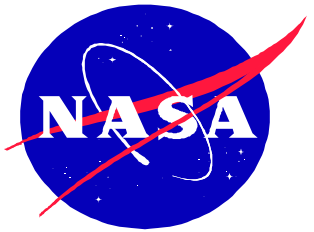


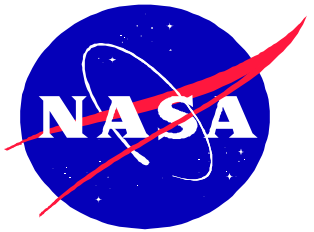
Model Checking for Software

Willem Visser
Charles Pecheur

RIACS / NASA Ames

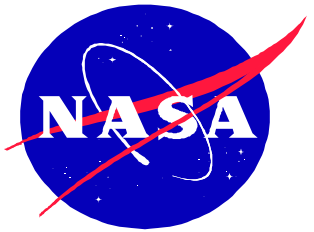
{wvisser,pecheur}@ptolemy.arc.nasa.gov

Part I
Overview



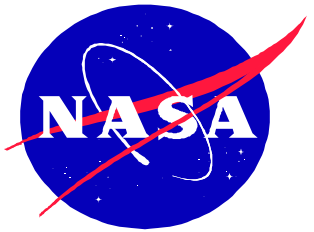
- Explicit State Model Checking
  - What is model checking?
  - Kripke structures
    - Describing the systems we want to check
  - Temporal logic
    - Describing the properties we want to check
  - Automata-theoretic model checking
  - State-explosion problem
    - What can we do?
- Model Checking Programs
  - A brief history of the field
  - Java PathFinder

Model Checking
The Intuition



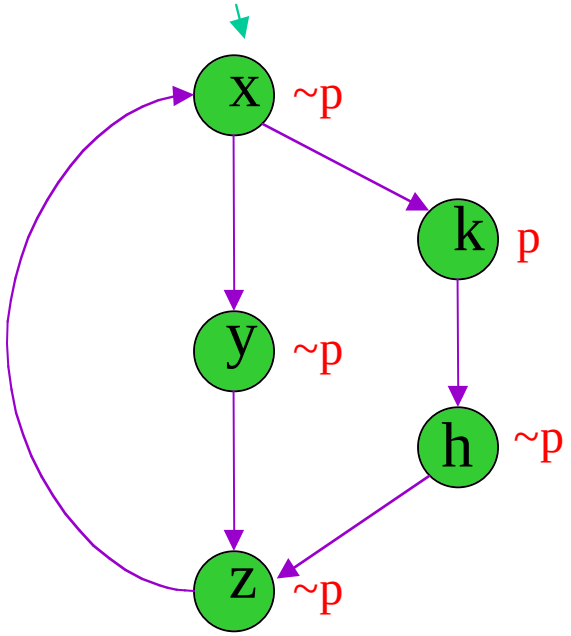
- Calculate whether a system satisfies a certain behavioral property:
  - Is the system deadlock free?
  - Whenever a packet is sent will it eventually be received?
- Testing?
  - Look at *all* possible behaviors of a system
- Automatic, if the system is finite-state
  - Potential for being a push-button technology
  - Almost no expert knowledge required
- How do we describe the system?
- How do we express the properties?

Kripke Structures



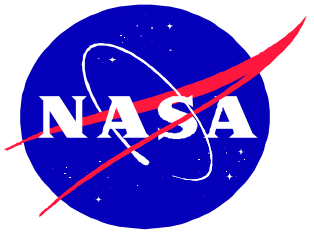
- $K = (props, S, R, s_0, L)$
  - *props* : (finite) set of atomic propositions
  - S : (finite) set of states
  - R : binary transitive relation (total)
  - s_0 : set of initial states
  - L : maps each state to the set of propositions true in the state
- Often $M = (S, R, L)$ with *props* and s_0 implicit

Example Kripke Structure



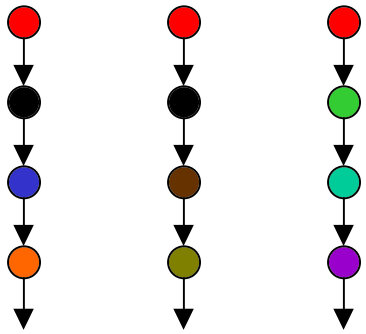
K = ({p,~p},{x,y,z,k,h},R,{x},L)

Property Specifications

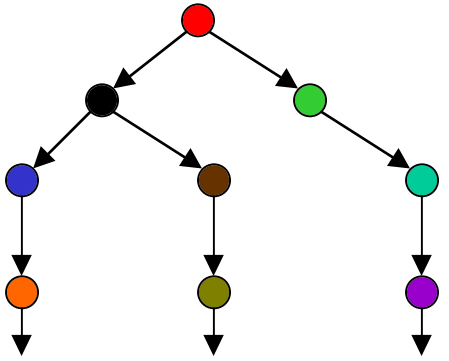


- Temporal Logic
  - Express properties of event orderings in time

- Linear Time
  - Every moment has a unique successor
  - Infinite sequences (words)
  - Linear Time Temporal Logic (LTL)



- Branching Time
  - Every moment has several successors
  - Infinite tree
  - Computation Tree Logic (CTL)



CTL*

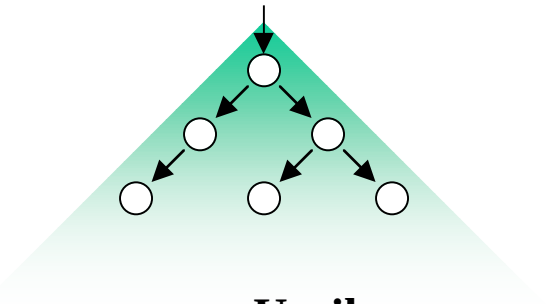
$$S ::= true \mid false \mid q \mid \sim q \mid S \vee S \mid S \wedge S \mid AP \mid EP$$

$$P ::= S \mid P \vee P \mid P \wedge P \mid XP \mid P\ U\ P \mid P\ V\ P$$

- *A* (for all) and *E* (there exists) are path quantifiers
- *X* (until), *U* (until) and *V* (release) are path operators
- CTL: Every path operator is preceded by a path quantifier
- LTL: Path formulas of the form *A*P where the state sub-formulas are atomic propositions
- *F*p = true *U* p - “eventually p” or “finally p”
- *G*p = false *V* p - “always p” or “globally p”

CTL

Computation Tree Logic (branching):
Consider the tree of possible executions



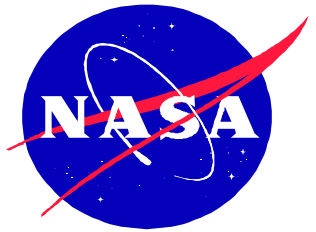
	... Next *p*	... Globally *p*	... Finally *p*	... *p* Until *q*
Always ...	**AX** *p*	**AG** *p*	**AF** *p*	**A**[*p* **U** *q*]
Sometimes ...	**EX** *p*	**EG** *p*	**EF** *p*	**E**[*p* **U** *q*]

In all states

duals

In some state

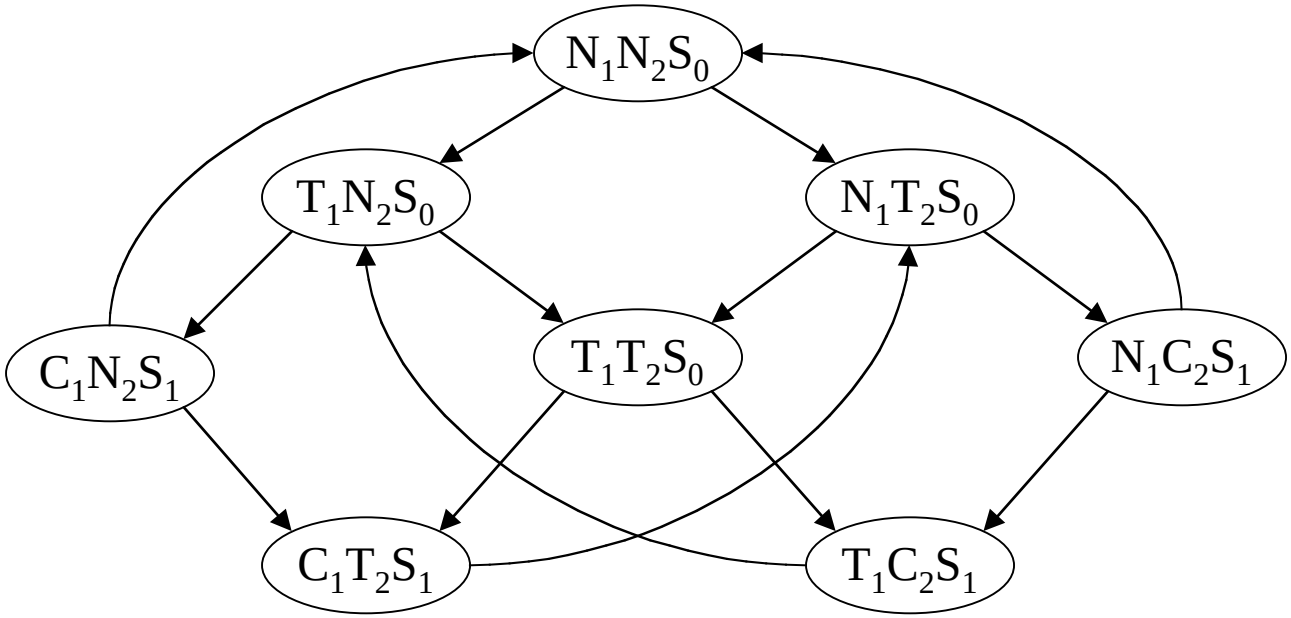
Mutual Exclusion Example



- Two process mutual exclusion with shared semaphore
- Each process has three states
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- Semaphore can be available (S_0) or taken (S_1)
- Initially both processes are in the Non-critical state and the semaphore is available --- $N_1\, N_2\, S_0$

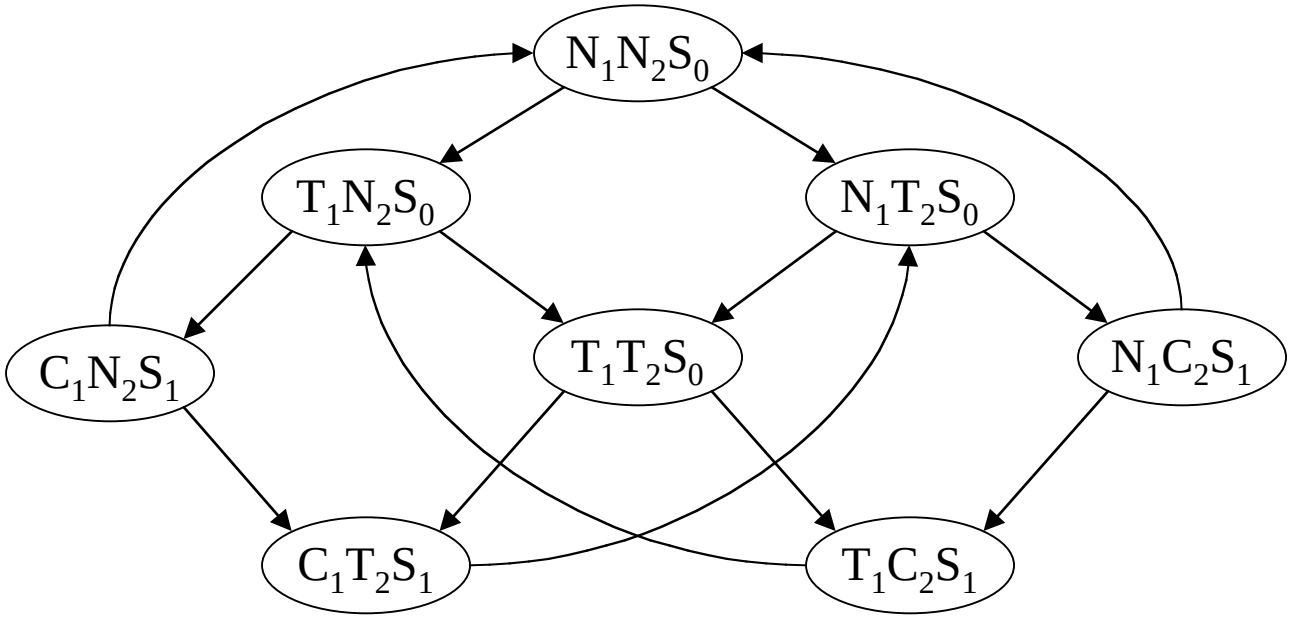
$$\begin{array}{lcl} N_1 & \rightarrow & T_1 \\ T_1 \wedge S_0 & \rightarrow & C_1 \wedge S_1 \\ C_1 & \rightarrow & N_1 \wedge S_0 \end{array} \quad \| \quad \begin{array}{lcl} N_2 & \rightarrow & T_2 \\ T_2 \wedge S_0 & \rightarrow & C_2 \wedge S_1 \\ C_2 & \rightarrow & N_2 \wedge S_0 \end{array}$$

Mutual Exclusion Example



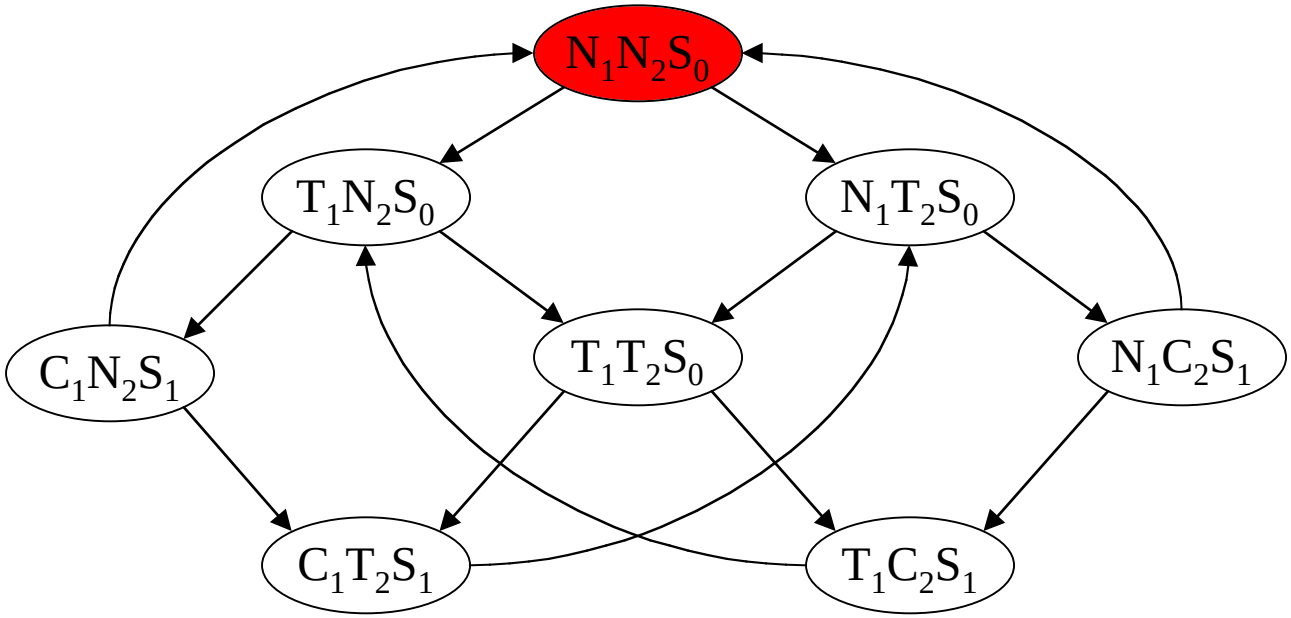
- Mutual Exclusion: K _ AG $\sim(C_1 \wedge C_2)$
- Response : K $\not\_$ AG $(T_1 \rightarrow$ AF $(C_1))$
- Reactive : K _ AG EF $(N_1 \wedge N_2 \wedge S_0)$

Mutual Exclusion Example



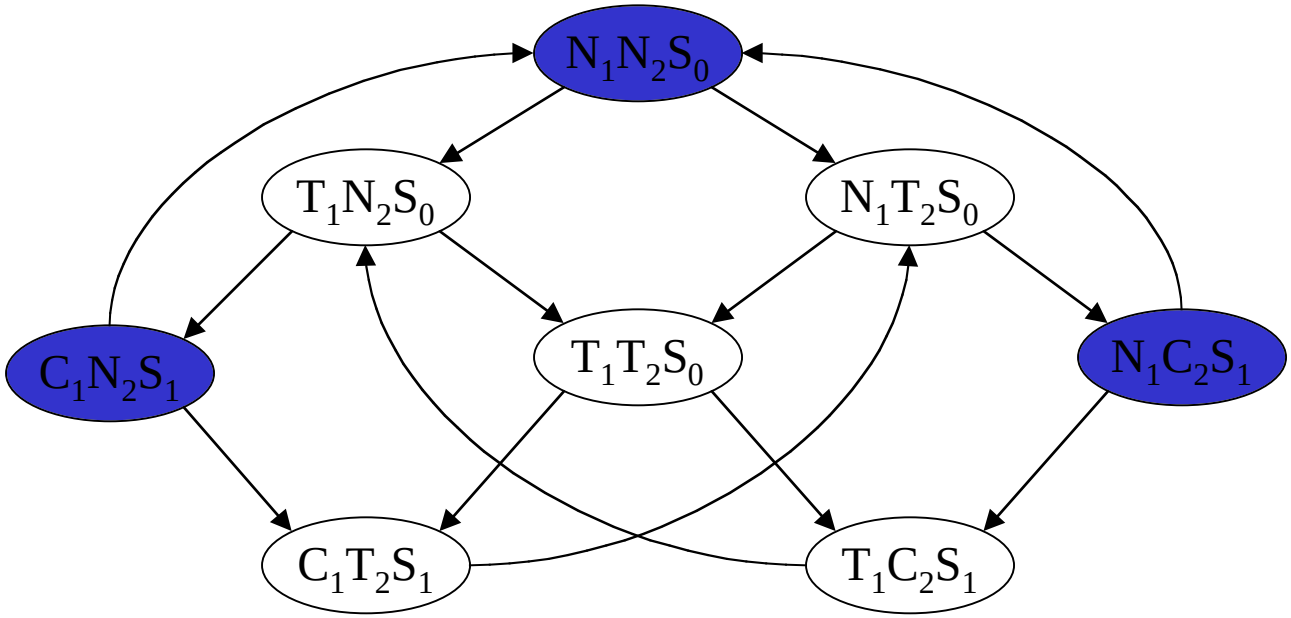
$$K \_ AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



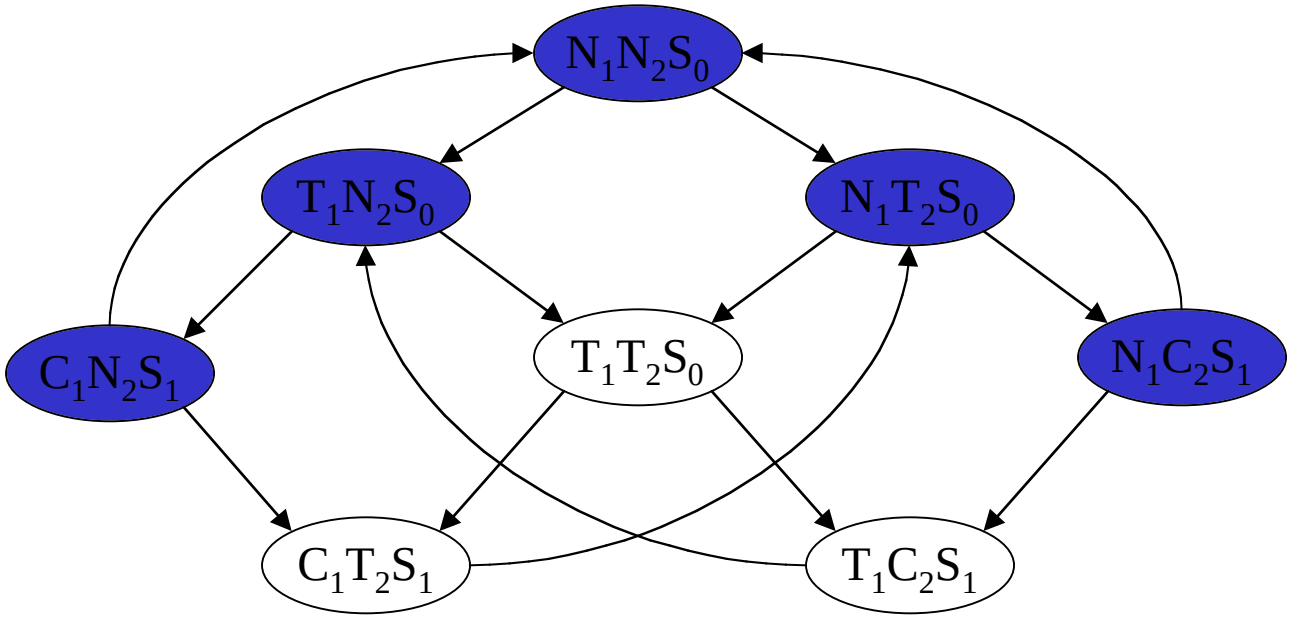
$$K\_AG\,EF\,(N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



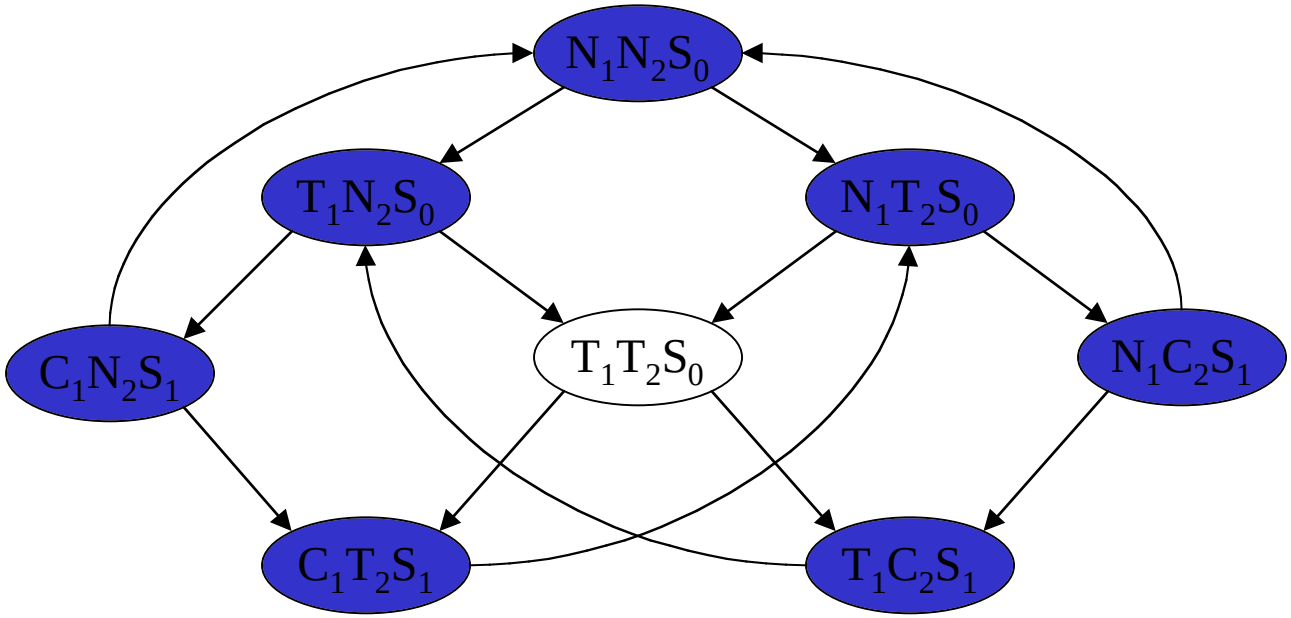
$$K \_ AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



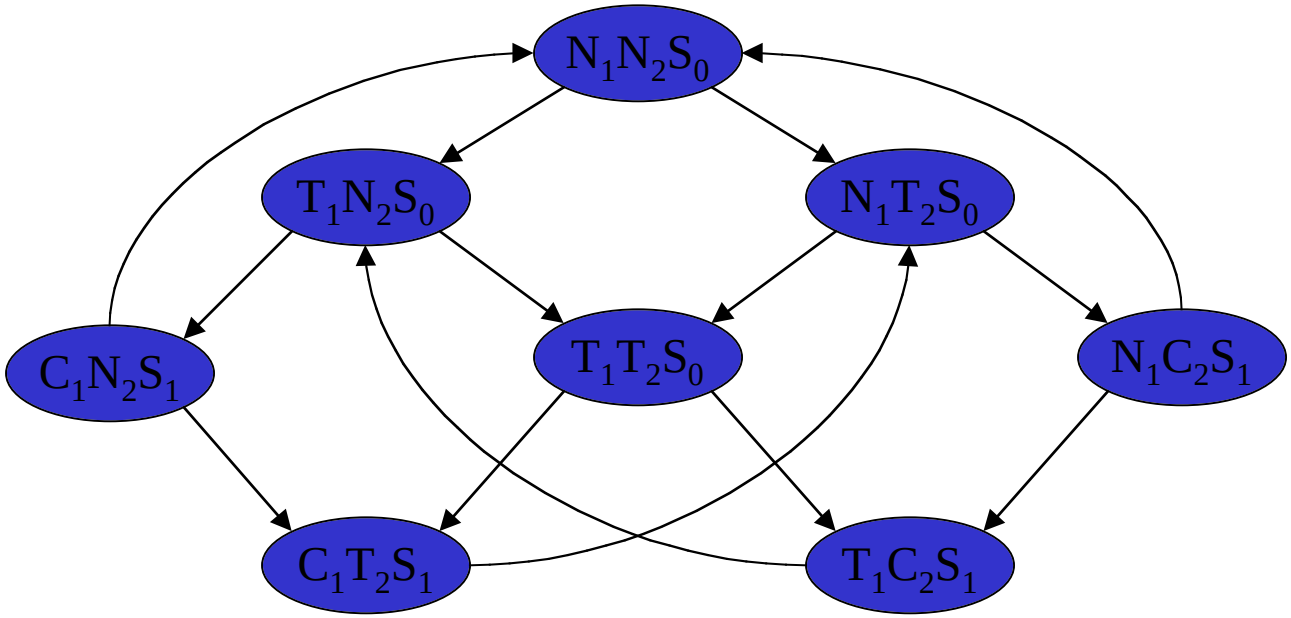
$$K \_ AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



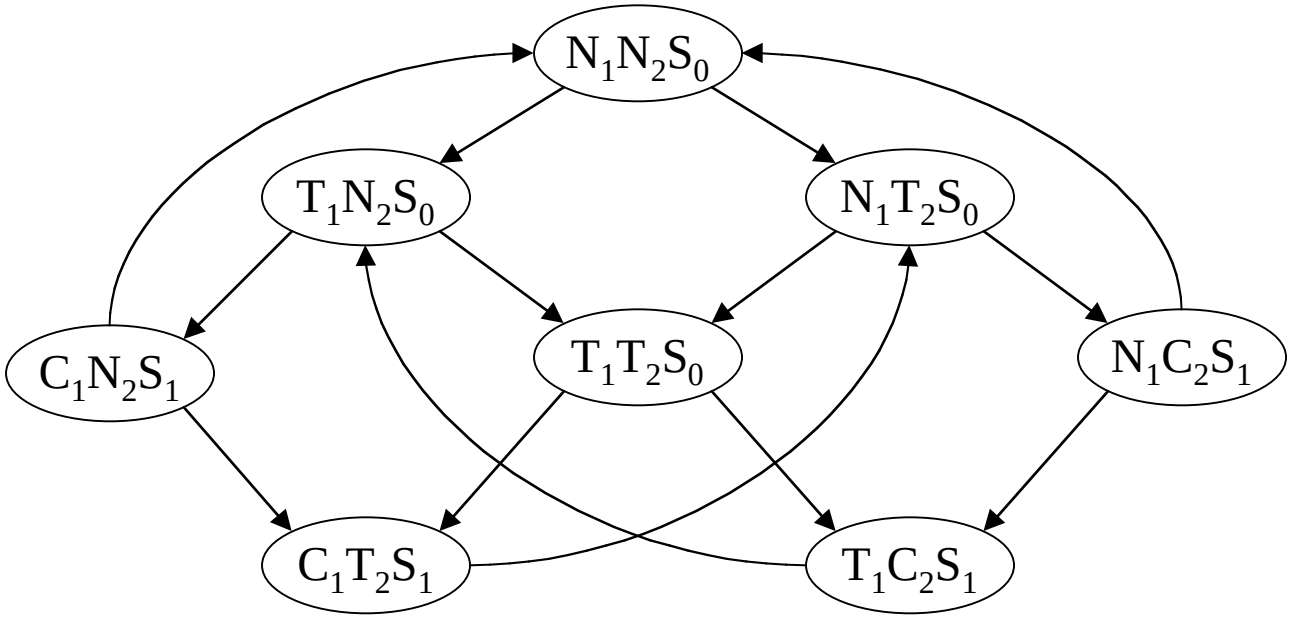
$$K \_ AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



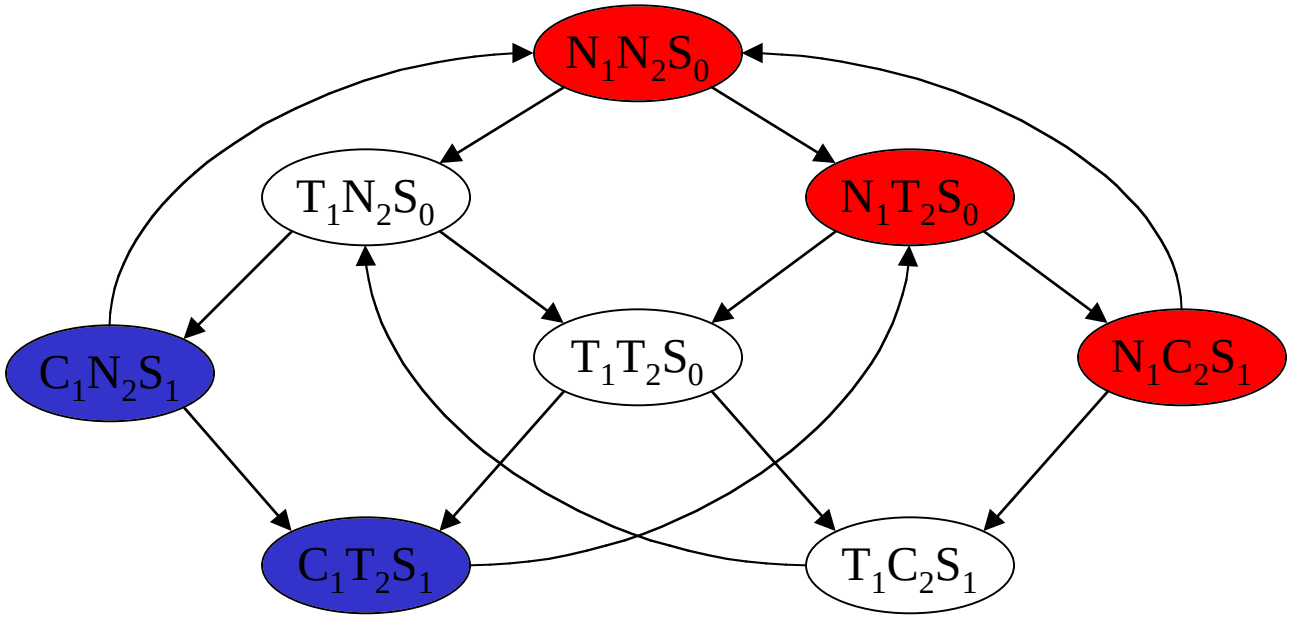
$$K \_ AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



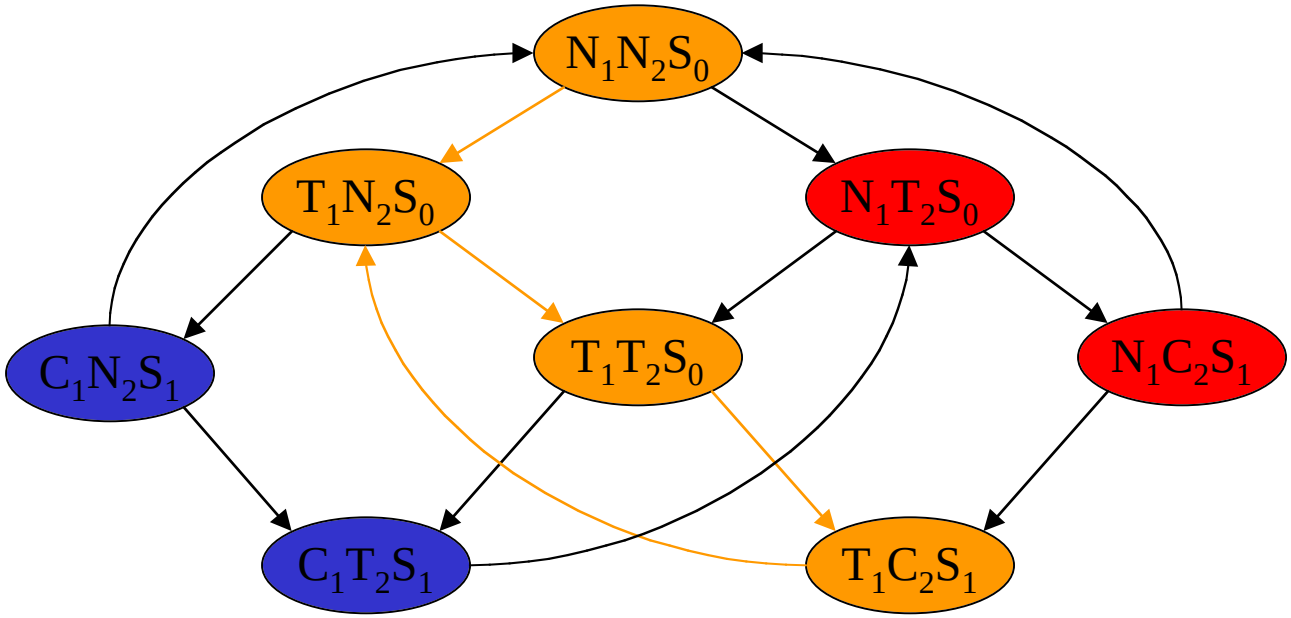
$$K \not\models AG\,(T_1 \rightarrow AF\,(C_1))$$

Mutual Exclusion Example



$$K \not\models AG\,(\sim T_1 \vee AF\,(C_1))$$

Mutual Exclusion Example



$$K \not\models AG(\sim T_1 \vee AF(C_1))$$

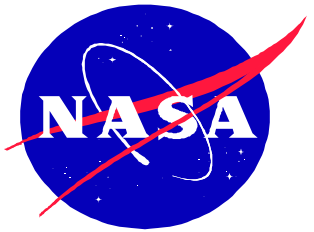
Model Checking

- Given a Kripke structure $M = (S,R,L)$ that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, find the set of states in S that satisfy f:

$$\{ s \in S \mid M,s \_ f \}$$

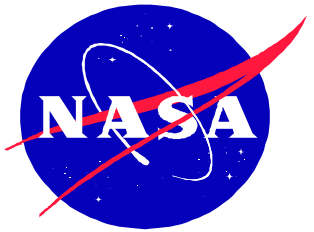
- Normally, some states of the concurrent system are designated as initial states. The system satisfies the specification provided all the initial states are in the set. We often write: K _ f

Model Checking Complexity K _ f



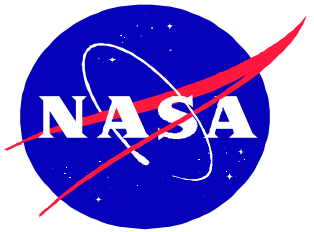
- CTL
  - $O(|K| * |f|)$
- LTL
  - $O(|K| * 2^{|f|})$
- But, for CTL the whole transition relation must be kept in memory!
  - Binary Decision Diagrams (BDDs) often allows the transition relation to be encoded efficiently
- The formulas are seldom very complex, hence |f| is not too troublesome.

Automata-Theoretic Model Checking



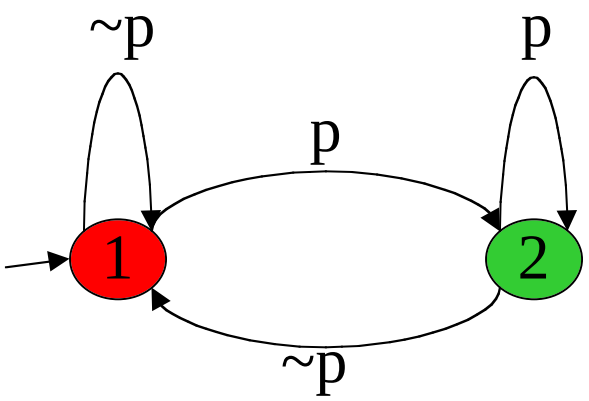
- Linear time temporal logic
  - Nondeterministic automata over infinite words
- Branching time temporal logic
  - Alternating automata over infinite trees
- Automata-theoretic LTL model checking
- Basic idea:
  - Translate both Kripke structure and LTL property into automata and show language containment
- See papers by Vardi and Wolper

Büchi Automata



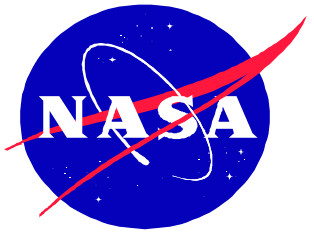
- Accepts infinite words
- $B = (\Sigma, S, \rho, s_0, F)$
  - Σ is a finite alphabet
  - S is a finite set of states
  - $\rho : S \times \Sigma \rightarrow 2^S$ is the transition function
  - $s_0 \in S$ is the initial state (or states)
  - $F \subseteq S$ is the set of accepting states
- Given an infinite word $\omega = a_0, a_1, \ldots$ over Σ then a *run* of B is the sequence $s_0, s_1, \ldots$ where $s_{i+1} \in \rho(s_i, a_i)$
- Let *inf*(π) be the set of states that occur infinitely often on the run π, then π is accepting *iff* *inf* $(\pi) \cap F \neq \emptyset$

Example Büchi Automaton



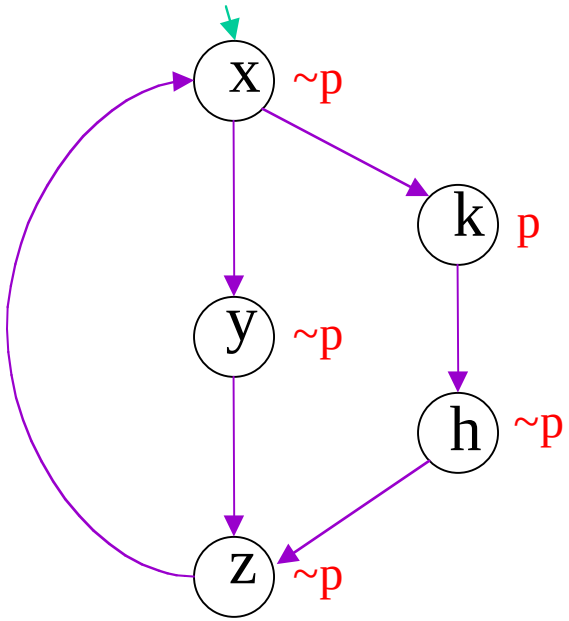
- B = ({{p},{~p}},{1,2}, ρ, 1, {2})
- Example accepting words:
  - $(12)^{\omega}$
  - 1112^{ω}
- Example rejecting word: 121212111^{ω}
- LTL property: GFp – “infinitely often p”

Kripke to Büchi Automaton

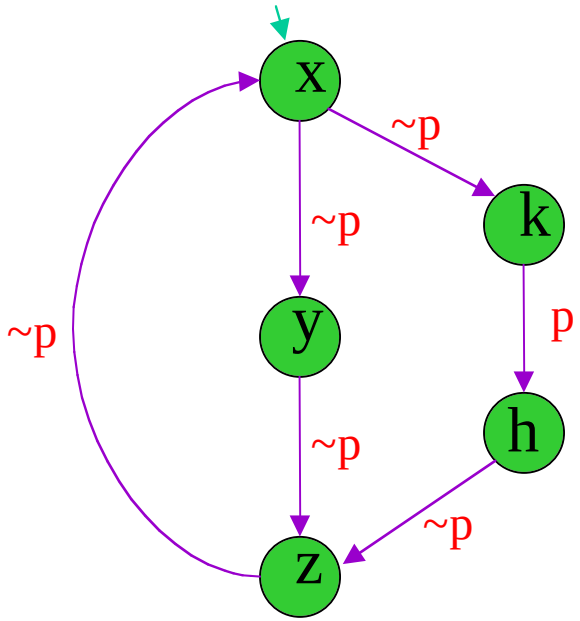


- $K = (props, S, R, s_0, L)$ can be viewed as
- $A_K = (2^{props}, S, \rho, s_0, S)$ where
  - $s_{i+1} \in \rho(s_i, a)$ *iff* $(s_i, s_{i+1}) \in R$ and $a = L(s)$
  - Note every state is in the accepting set, hence all runs of the automaton is accepting
  - The language of the automaton, $\mathbf{L}(A_K)$, is the set of all behaviors of K

Kripke to Büchi Example

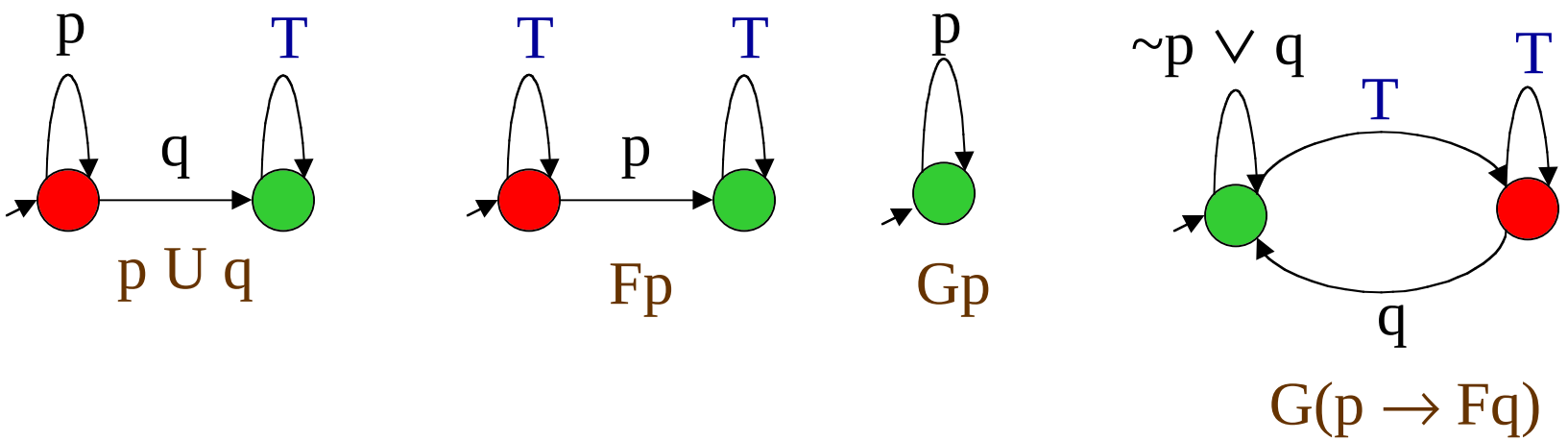


Kripke to Büchi Example



Translating LTL Formulas to Büchi Automata

- Exponential in the length of the formula
  - Many heuristics optimizations are used
  - Multitude of papers: CAV, LICS, etc.

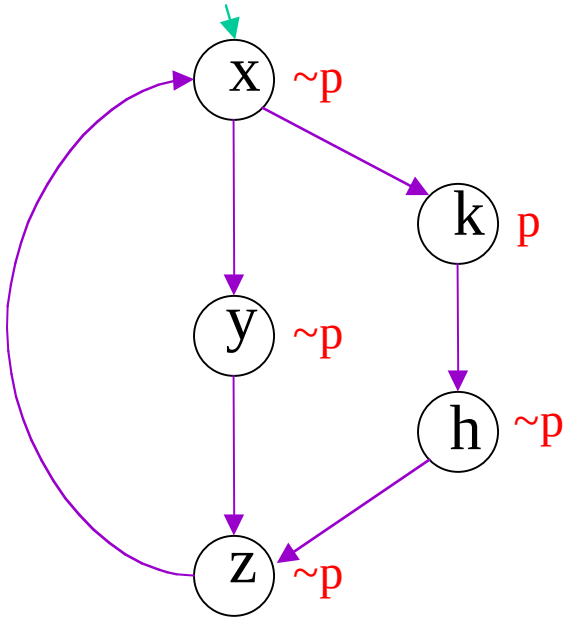


Model Checking with Büchi Automata

- K _ f
- Translate K and f to Büchi Automata
- Language containment
  - $\mathbf{L}(A_K) \subseteq \mathbf{L}(\underline{A_f})$
  - $\mathbf{L}(A_K) \cap \mathbf{L}(A_f) = \emptyset$
  - $\mathbf{L}(\overline{A_f}) = \mathbf{L}(A_{\sim f})$ and $\mathbf{L}(A_K \times A_{\sim f}) = \mathbf{L}(A_K) \cap \mathbf{L}(A_{\sim f})$
- Algorithm
  - Negate formula f and create $A_{\sim f}$
  - Construct the product $A_{K,\sim f} = K \times A_{\sim f}$
  - If $\mathbf{L}(A_{K,\sim f}) = \emptyset$ report YES else report NO

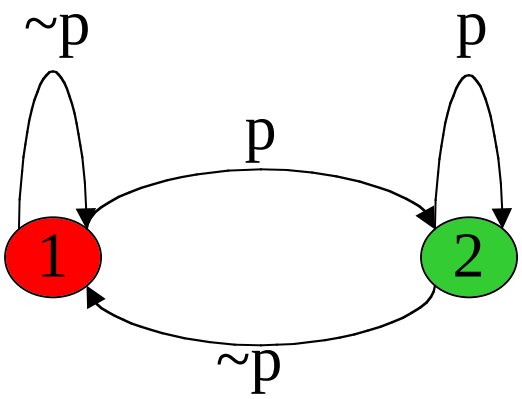
Model Checking Example

- K _ AFG~p
  - For all paths from some moment onwards p is always false
- Where K is given by



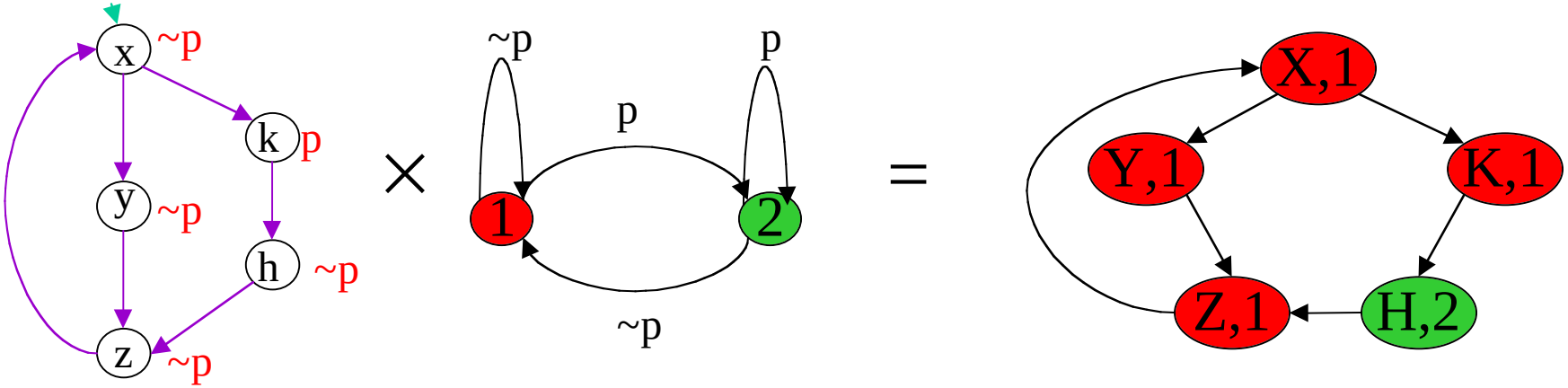
Step 1

- Negate FG~p
  - GFp
- Construct Büchi Automaton for GFp

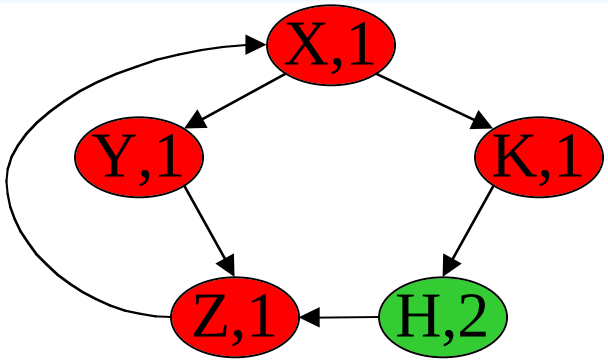


Step 2

- Construct the product automaton

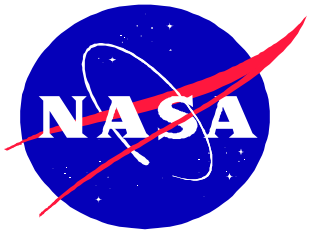


Step 3



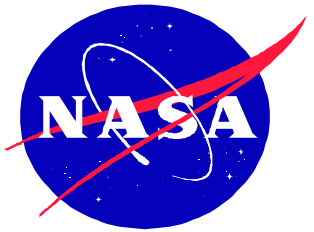
- Check if the language is empty
- It is nonempty since there is a cycle through an accepting state, hence K _/AFG~p
  - $(xkhz)^{\omega}$ is an accepting run
- The accepting run is also a counter-example to the property being true

Checking Nonemptiness



- A Büchi automaton accepts some word *iff* there exists an accepting state reachable from the initial state and from itself
- Can be checked in linear time
- Model Checking complexity for LTL
  - $O(|K| * 2^{|f|})$

Efficient Nonemptiness Checking



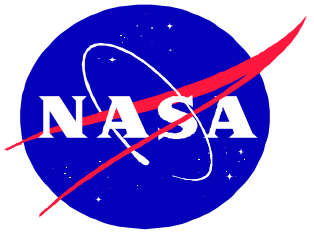

```
Dfs (state s)
    Add (s,0) to VisitedStates;
    FOR each successor t of s DO
        IF (t,0) ∉ VisitedStates THEN Dfs(t) END
    END
    IF s ∈ F THEN seed := s; 2Dfs(s) END
END

2Dfs (state s)
    Add (s,1) to VisitedStates;
    FOR each successor t of s DO
        IF (t,1) ∉ VisitedStates THEN 2Dfs(t) END
        ELSEIF t = seed THEN report nonempty END
    END
END
```

Efficiency

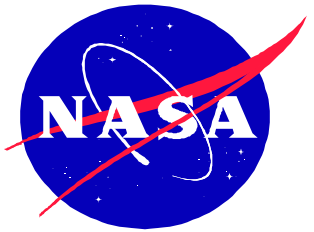
- VisitedStates as HashTable
- Change Recursion to Iteration
- Generate successor states on-the-fly

SPIN Model Checker



- Automata based model checker
  - Efficient nonemptiness algorithm
- Translates LTL formula to Büchi automaton
- Kripke structures are described as "programs" in the PROMELA language
  - Kripke structure is generated on-the-fly during nonemptiness checking
- http://netlib.bell-labs.com/netlib/spin/whatispin.html
  - Relevant theoretical papers can be found here

State-Explosion?

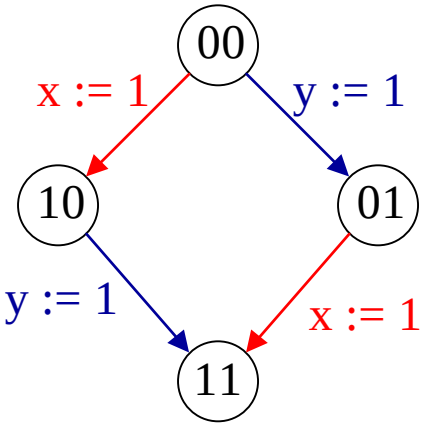


- n concurrent processes with m states each
  - Has m^n states
  - Worst-case, an on-the-fly model checker has to enumerate all of them
- What can we do to reduce m^n ?
  - Reduce m
    - Abstraction
  - Reduce the effect of n
    - Partial-order reductions
  - Reduce n
    - Symmetry reductions

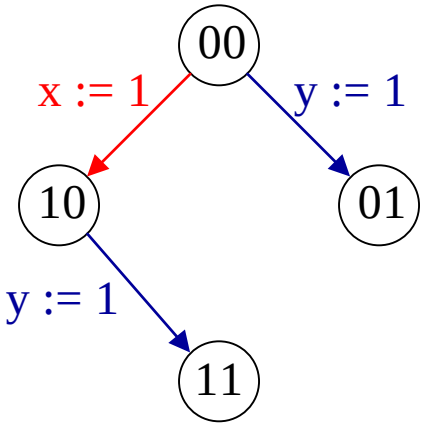
We'll consider these 2 here

Partial-Order Reductions

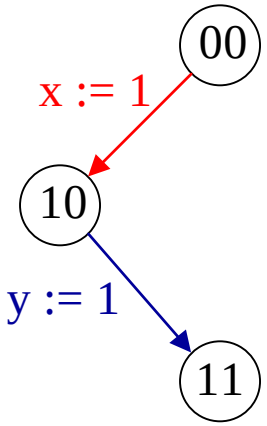
- Reduce the number of interleavings of independent concurrent transitions
- x := 1 || y := 1 where initially x = y = 0



No Reductions

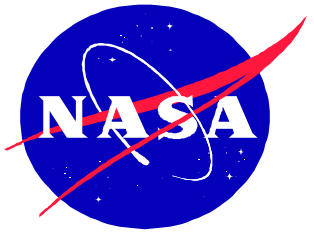


Transitions Reduced



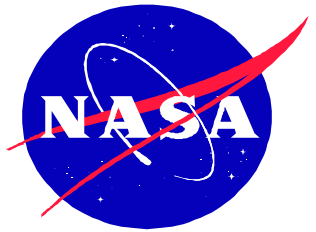
States Reduced

Basic Ideas



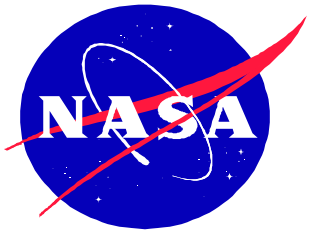
- Independence
  - Independent transitions cannot disable nor enable each other
  - Enabled independent transitions are commutative
- Partial-order reductions only apply during the on-the-fly construction of the Kripke structure
- Based on a *selective search* principle
  - Compute a subset of enabled transitions in a state to execute
- Sleep sets (Reduce transitions)
- Persistent sets (Reduce states)

Persistent Set Reductions



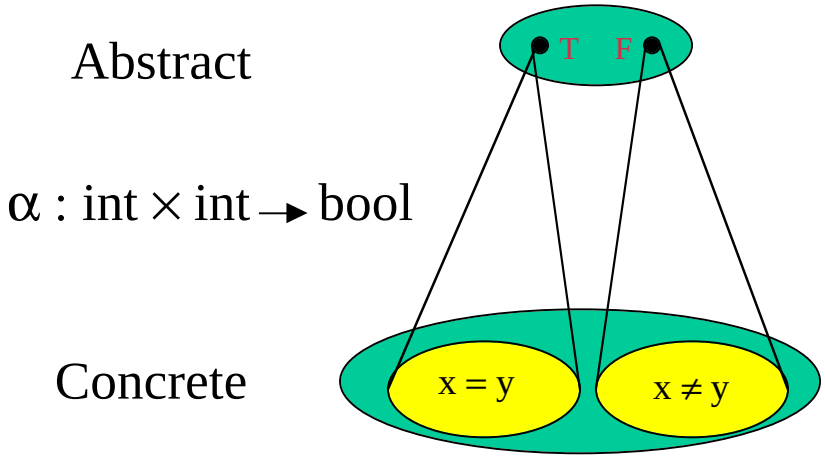
- A subset of the enabled transitions is called *persistent*, when any transition can be executed from outside the set and it will not interact or affect the transitions inside the set
  - Use the static structure of the system to determine what goes into the persistent set
  - Note, all enabled transitions are trivially persistent
- Only execute transitions in the persistent set
- Persistent set algorithm is used within SPIN
- See papers by Godefroid and Peled

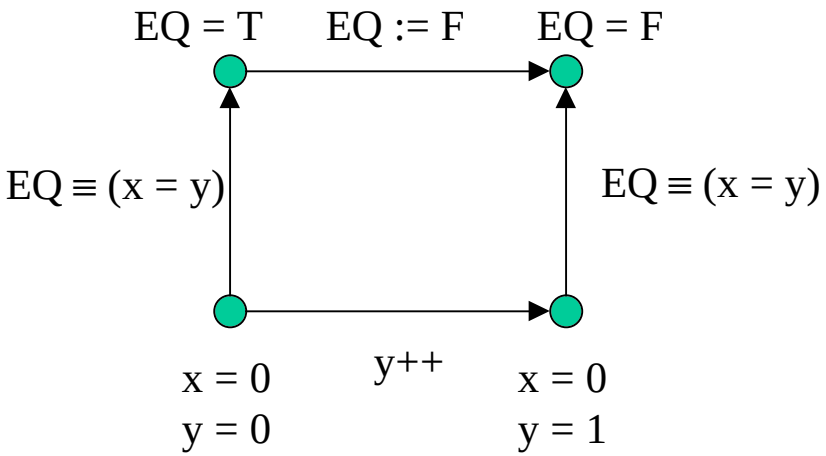
Abstraction



- Type based abstractions
  - Abstract Interpretation
  - Replace integer variable with odd-even range
  - Or Signs abstraction: negative,zero,positive
  - Replace all operations on the concrete variable with corresponding abstract operations
    - add(pos,pos) = pos
    - subtract(pos,pos) = negative | zero | pos
    - eq(pos,pos) = true | false
- Predicate Abstraction (Graf, Saïdi see also Uribe)
  - Create abstract state-space w.r.t. set of predicates defined in concrete system

Predicate Abstraction





- Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate
- Create abstract state-graph during model checking, or,
- Create an abstract transition system before model checking

Example
Predicate Abstraction

Predicate: $B \equiv (x = y)$

Concrete Statement

$y := y + 1$

Abstract Statement

Step 1: Calculate pre-images

$\{x = y + 1\}\ y := y + 1\ \{x = y\}$

$\{x \neq y + 1\}\ y := y + 1\ \{x \neq y\}$

Step 2: Rewrite in terms of predicates

$\{x = y + 1\}\ y := y + 1\ \{B\}$

$\{B\}\ y := y + 1\ \{\sim B\}$

Step 2a: Use Decision Procedures

$x = y \rightarrow x = y + 1$ $\quad x = y \rightarrow x \neq y + 1$

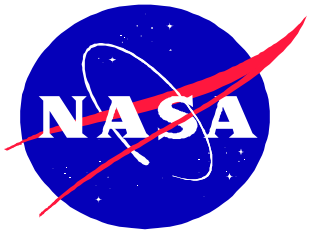
$x \neq y \rightarrow x = y + 1$ $\quad x \neq y \rightarrow x \neq y + 1$

Step 3: Abstract Code

IF B THEN B := false

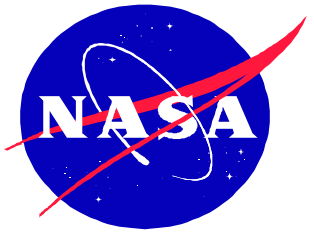
ELSE B := true | false

The Story so Far



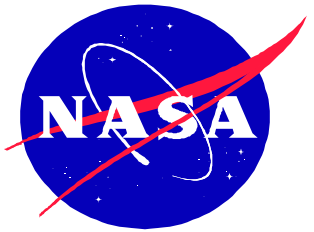
- Explicit State Model Checking
  - Kripke structures
    - Describing the systems we want to check
  - Temporal logic
    - Describing the properties we want to check
  - Automata-theoretic model checking
  - State-explosion problem
    - Partial-order reductions
    - Predicate Abstraction
- Next - Model Checking Programs
  - A brief history of the field
  - Java PathFinder

Model Checking Programs



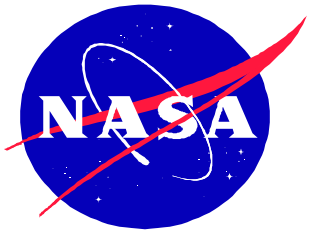
- Model checking usually applied to designs
  - Some errors get introduced after designs
  - Design errors are missed due to lack of detail
  - Sometimes there is no design
- Can model checking find errors in real programs?
  - Yes, many examples in the literature
- Can model checkers be used by programmers?
  - Only if it takes real programs as input

Main Issues



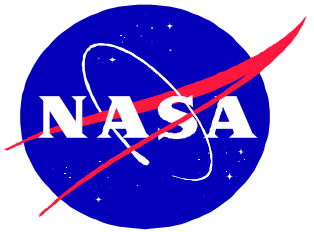
- Memory
  - Explicit-state model checking's Achilles heel
  - State of a software system can be complex
  - Require efficient encoding of state, or,
  - State-less model checking
- Input notation not supported
  - Translate to existing notation
  - Custom-made model checker
- State-space Explosion

State-less Model Checking



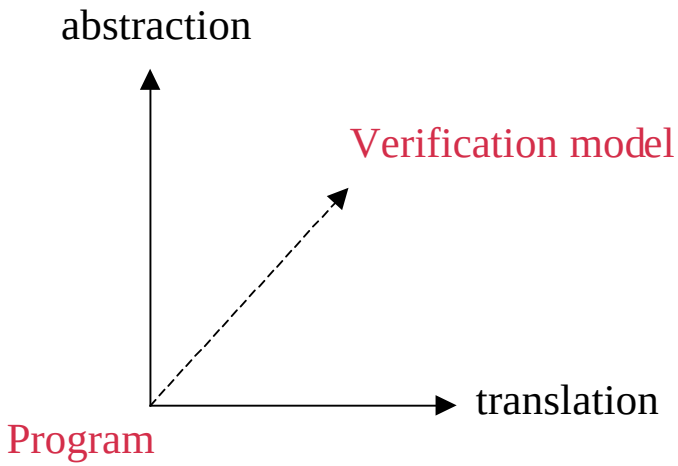
- Must limit search-depth to ensure termination
- Based on partial-order reduction techniques
- Annotate code to allow verifier to detect "important" transitions
- Examples include
  - VeriSoft
    - http://www1.bell-labs.com/project/verisoft/
  - Rivet
    - http://sdg.lcs.mit.edu/rivet.html

Traditional Model Checking



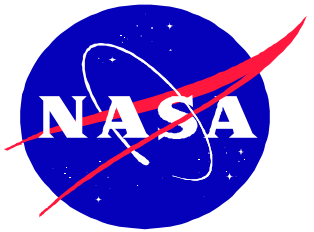
- Translation-based using existing model checker
  - Hand-translation
  - Semi-automatic translation
  - Fully automatic translation
- Custom-made model checker
  - Fully automatic translation
  - More flexible

Hand-Translation



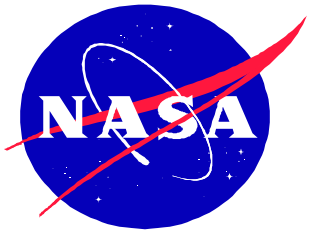
- Hand translation of program to model checker's input notation
- "Meat-axe" approach to abstraction
- Labor intensive and error-prone

Hand-Translation Examples



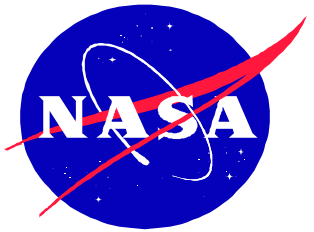
- Remote Agent – Havelund,Penix,Lowry 1997
  - http://ase.arc.nasa.gov/havelund
  - Translation from Lisp to Promela (most effort)
  - Heavy abstraction
  - 3 man months
- DEOS – Penix *et al.* 1998/1999
  - http://ase.arc.nasa.gov/visser
  - C++ to Promela (most effort in environment)
  - Limited abstraction - programmers produced sliced system
  - 3 man months

Semi-Automatic Translation



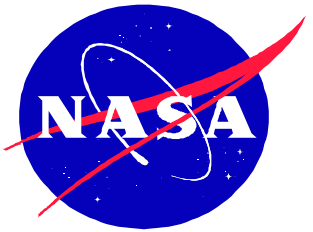
- Table-driven translation and abstraction
  - Feaver system by Gerard Holzmann
  - User specifies code fragments in C and how to translate them to Promela (SPIN)
  - Translation is then automatic
  - Found 75 errors in Lucent's PathStar system
  - http://cm.bell-labs.com/cm/cs/who/gerard/
- Advantages
  - Can be reused when program changes
  - Works well for programs with long development and only local changes

Fully Automatic Translation



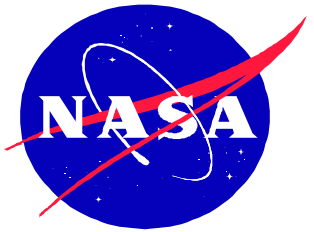
- Advantage
  - No human intervention required
- Disadvantage
  - Limited by capabilities of target system
- Examples
  - Java PathFinder 1- http://ase.arc.nasa.gov/havelund/jpf.html
    - Translates from Java to Promela (Spin)
  - JCAT - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml
    - Translates from Java to Promela (or dSpin)
  - Bandera - http://www.cis.ksu.edu/santos/bandera/
    - Translates from Java bytecode to Promela, SMV or dSpin

Custom-made Model Checkers



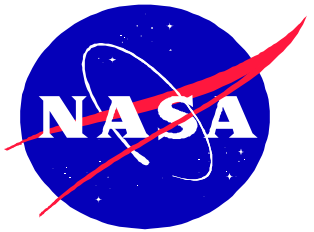
- Allows efficient model checking
  - Often no translation is required
  - Algorithms can be tailored
- Translation-based approaches
  - dSpin
    - Spin extended with dynamic constructs
    - Essentially a C model checker
    - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml
  - Java Model Checker (from Stanford)
    - Translates Java bytecode to SAL language
    - Custom-made SAL model checker
    - http://sprout.stanford.edu/uli/

Java PathFinder 2



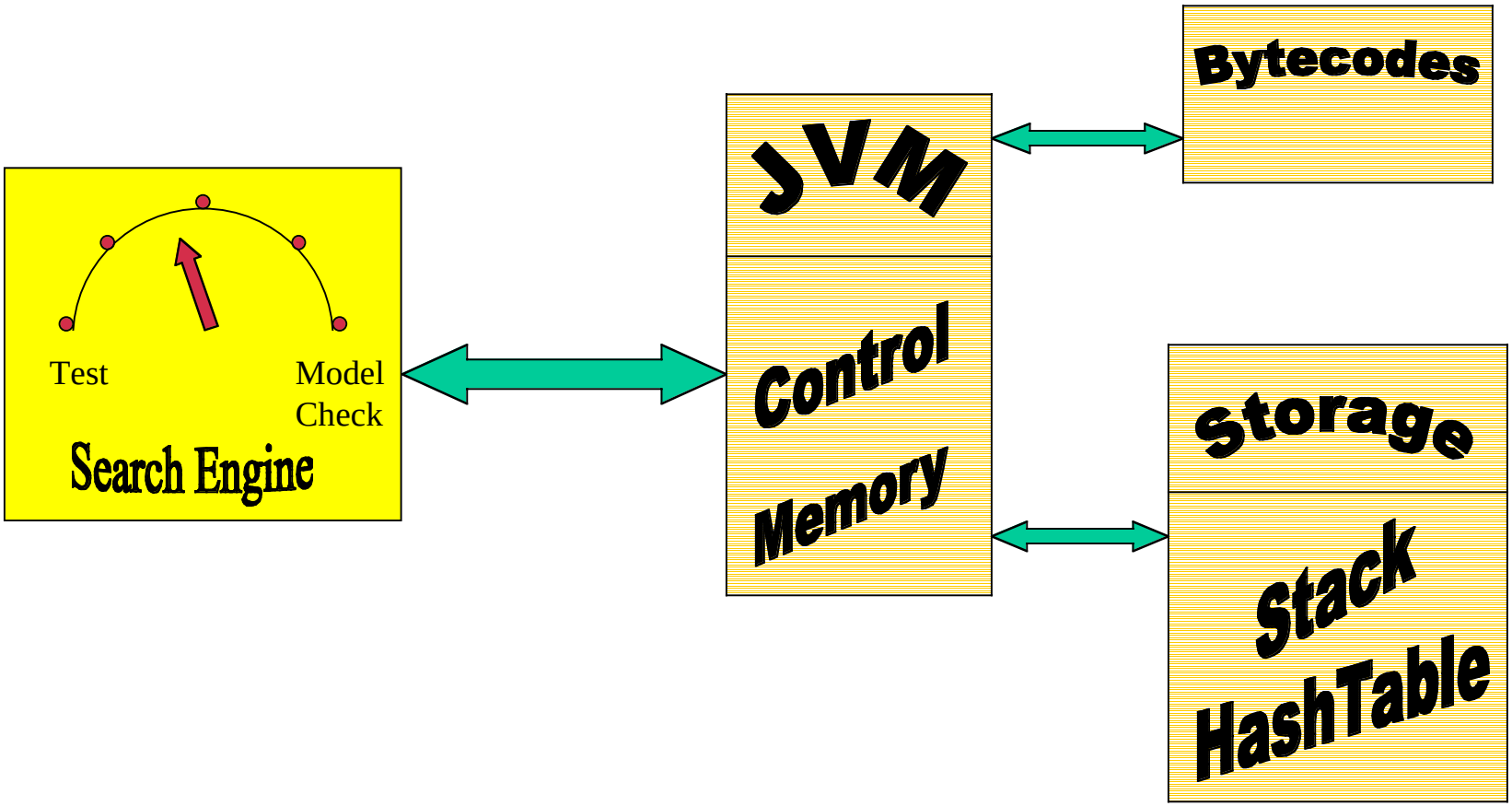
- Based on new Java Virtual Machine
  - Handle all of Java, since it works with bytecodes
- Written in Java
  - 1 month to develop version with only integers
- Efficient encoding of states
  - Complex states are translated to integer vector
  - Garbage collection
  - Canonical heap representation
- http://ase.arc.nasa.gov/jpf

JPF2 Core Decisions

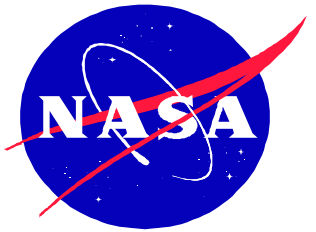


- Explicit-state model checking
- Build own Java Virtual Machine
  - Emphasis on memory management not speed
  - Bytecode level assures language coverage
- Modular design to allow flexible system
  - Different search algorithms from testing to model checking

JPF2 Structure

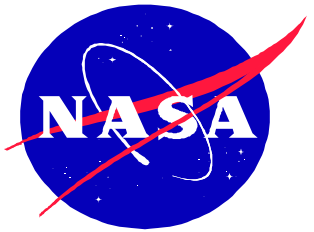


Our JVM



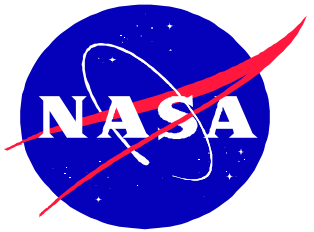
- Written in Java
  - *Java-in-Java* at SUN and *Rivet* at MIT
- Use *JavaClass* package
  - Class loading, Internal class file structure, etc.
- Initial implementation took 1 month!
- State encoded in complex data-structures
- Exploit Java in Java at every opportunity

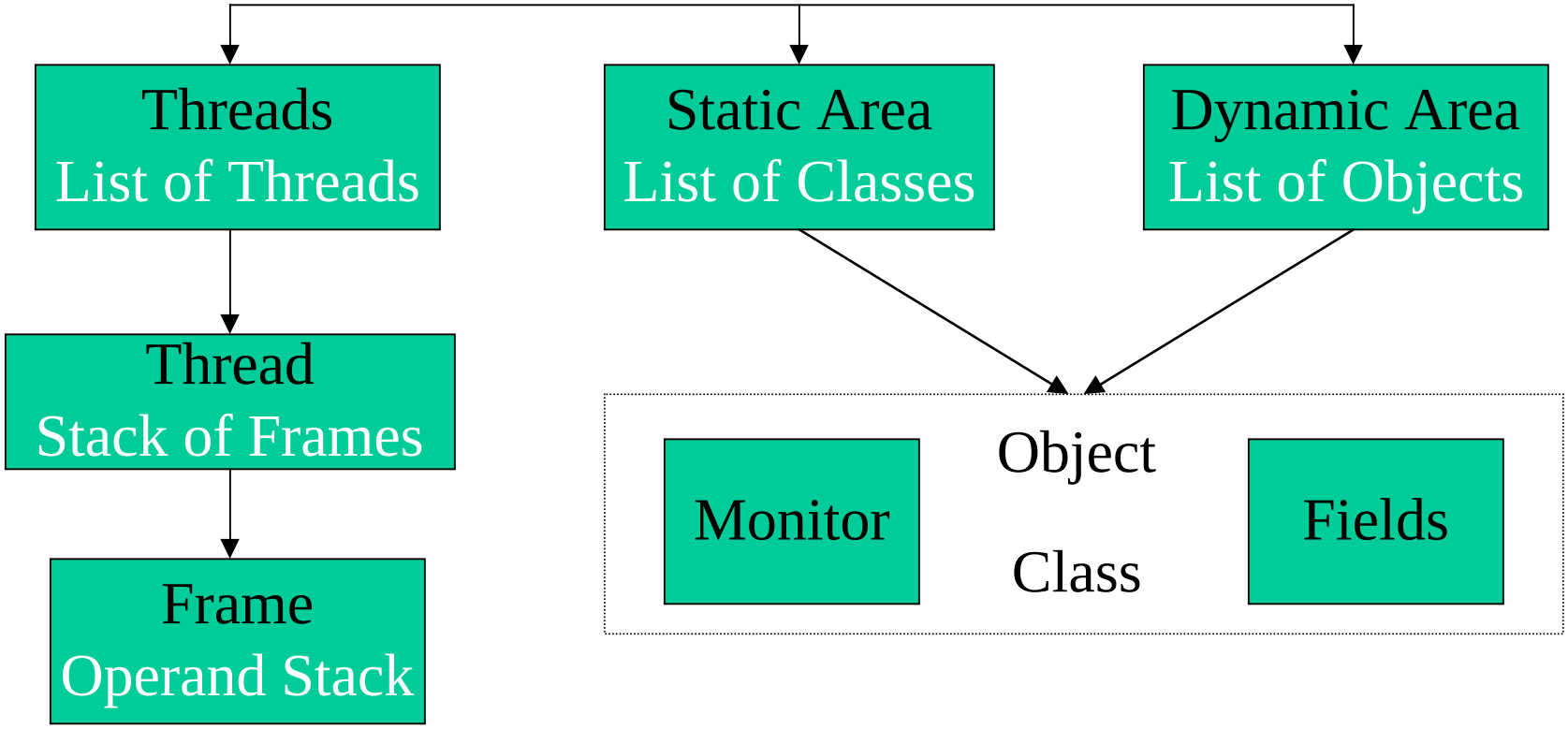
Memory Management



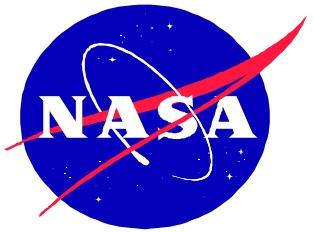
- SystemState
  - KernelState, i.e. JVM state
  - Scheduler, used for exploring all paths
- SystemState goes on Stack to allow backtracking
  - Use “clone” operation to store states on the stack
  - It can be slow
- KernelState goes in HashTable to record states
  - Generic part of system, stores byte-vectors
  - Use “Pools” to reduce complex state to byte-vector

JVM State



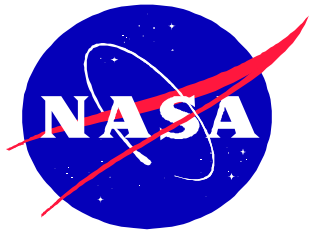


Reducing State Size



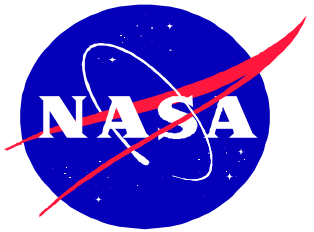
- Use intermediate tables/pools for each major class
  - Threads, Objects, Fields, Monitors
- Each time an object of the class must be stored, see if it is in the table, if so return the index, else insert it into next open slot and return the index
- Works well if tables don't become to big
- Optimization
  - Only calculate index when object has changed.

Reducing the State Space



- Partial-order reductions
  - Vital for efficient explicit-state model checking
  - Must be able to identify independent transitions
    - Static analysis
- Abstraction
  - Under-approximations
    - Slicing, i.e. a cultured “meat-axe”
  - Over-approximations
    - Predicate abstraction
    - Type-based abstraction

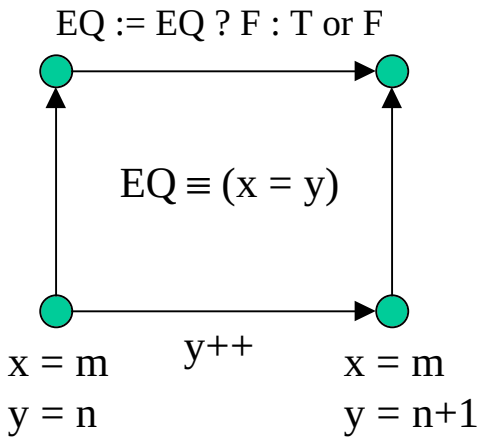
Slicing in JPF



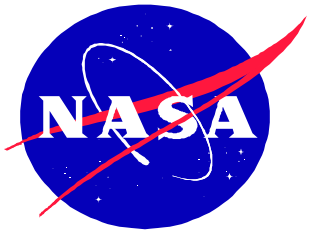
- JPF uses Bandera's slicer
- Bandera slices w.r.t.
  - Deadlock - i.e. communication statements
  - Variables occurring in temporal properties
  - Variables participating in race-violations
    - Used with JPF's runtime analysis
- More examples of slicing for model checking
  - Slicing for Promela (Millet and Teitelbaum)
    - http://netlib.bell-labs.com/netlib/spin/ws98/program.html
  - Slicing for Hardware Description Languages (Shankar *et al.*)
    - http://www.cs.wisc.edu/~reps/

JPF Abstraction Technique

- Find abstraction mapping (α) by user guidance
- Use decision procedures to automatically compute *abstract interpretation* of concrete transitions
- Validity checking of pre-images
- Over approximation with nondeterminism

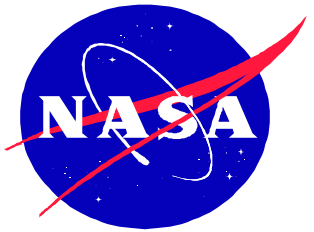


JPF's Java Abstraction



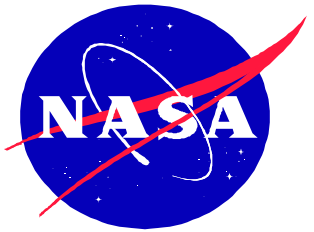
- Annotations used to indicate abstractions
  - `Abstract.remove(x);`
    `Abstract.remove(y);`
    `Abstract.addBoolean("EQ", x==y);`
- Tool generates abstract Java program
  - Using Stanford Validity Checker (SVC)
  - JVM is extended with nondeterminism to handle over approximation
- Abstractions can be local to a class or global across multiple classes
  - `Abstract.addBoolean("EQ", A.x==B.y);`
  - Dynamic predicate abstraction, since it works across instances

Runtime Analysis



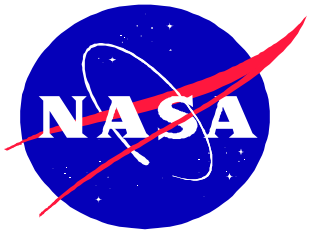
- Execute program once, and accumulate information to be analyzed
- Analysis can reveal an error potential although an error did not occur during the run
  - Looking for “footprints” of errors
- Data race violations with Eraser algorithm
- Lock order violations that lead to deadlock

Example Usage



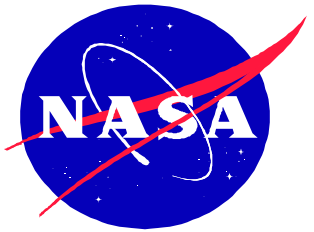
- Deep-Space 1 software deadlocked in flight
- Create a Java program from the Lisp
- Too large to model check (10^{60} states)
- Do race analysis and find violation
- Create slice (with Bandera) w.r.t. variable for which race violation occurred
- Model Check slice and find error (instantly)

Conclusions



- Low-hanging fruit principle
  - Errors always obvious (in hindsight!)
  - Model checkers are good at finding obvious errors
- Combine many different techniques
  - Abstraction, slicing, runtime analysis, etc.
- Current work
  - Adding property specification language
  - Finding “real” counter-examples during abstraction
  - Defining “environments” for Java programs
- htttp://ase.arc.nasa.gov/jpf

Part II
Symbolic Model Checking

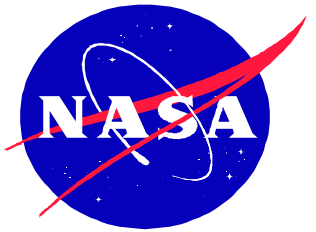


- Principles
  - BDDs
  - Symbolic MC algorithm
- Tools: *SMV*
  - Principles
  - Language
  - Variants

- Applications in Software
  - Model-based autonomy: *Livingstone*
  - Robot control: *TCL*

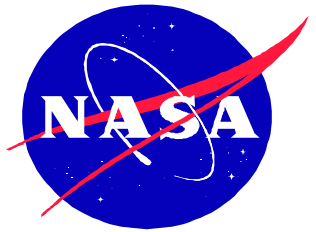
Based on material from:

- *Edmund Clarke*
- *Marius Minea*
- *Reid Simmons*



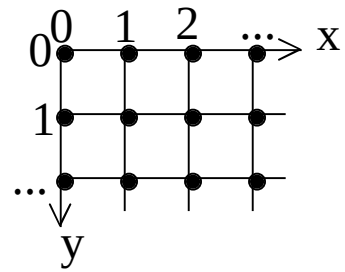
Symbolic Model Checking Principles

What is it?



Instead of considering each individual state,

Symbolic model checking...

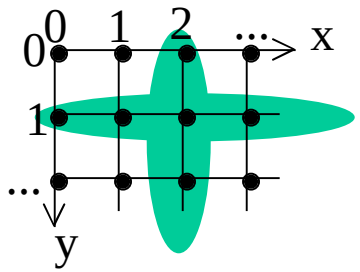


What is it?

Instead of considering each individual state,

Symbolic model checking...

- Manipulates sets of states,

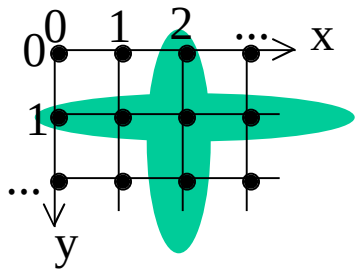


What is it?

Instead of considering each individual state,

Symbolic model checking...

- Manipulates sets of states,



- Represented as boolean formulas,

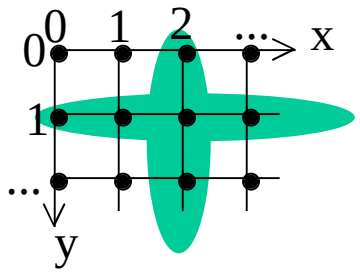
$x=2 \vee y=1$

What is it?

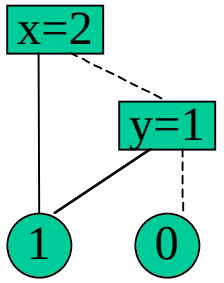
Instead of considering each individual state,

Symbolic model checking...

- Manipulates sets of states,
- Represented as boolean formulas,
- Encoded as binary decision diagrams.



$x=2 \vee y=1$

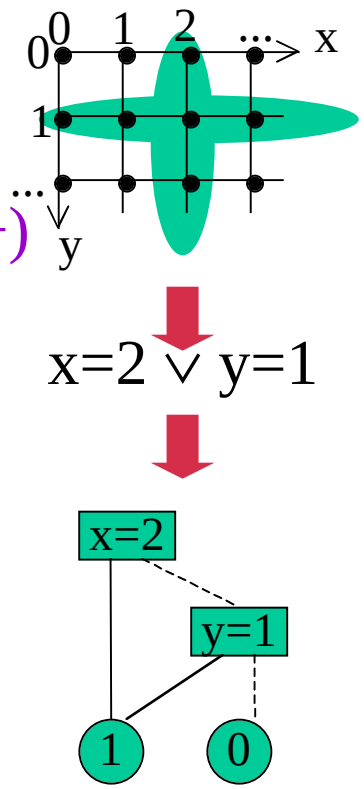


What is it?

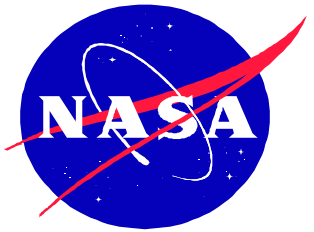
Instead of considering each individual state,

Symbolic model checking...

- Manipulates sets of states,
  - Can handle very large state spaces (10^{50} +)
- Represented as boolean formulas,
  - Suited for boolean/abstract models
- Encoded as binary decision diagrams.
  - The limit is BDD size (hard to control)



Boolean Functions



- Represent a state as boolean variables

$$s = b_1, ..., b_n$$

Non-boolean variables => use boolean encoding

- A set of states as a boolean function

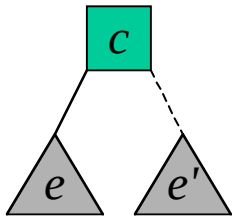
$$s \text{ in } S \text{ iff } f(b_1, ..., b_n) = 1$$

- A transition relation as a boolean function over two states

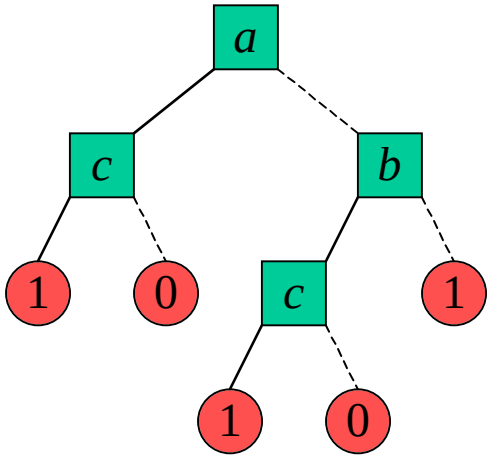
$$s \rightarrow s' \text{ iff } f(b_1, ..., b_n, b'_1, ..., b'_n) = 1$$

Binary Decision Trees

- Encoding for boolean functions
- Notational convention:



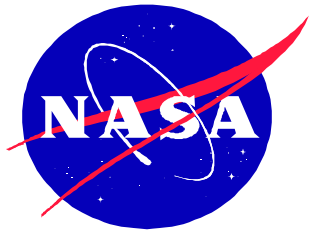
= **if** *c* **then** *e* **else** *e'*
= (*c* ? *e* : *e'*)



- Always exists
  but not unique

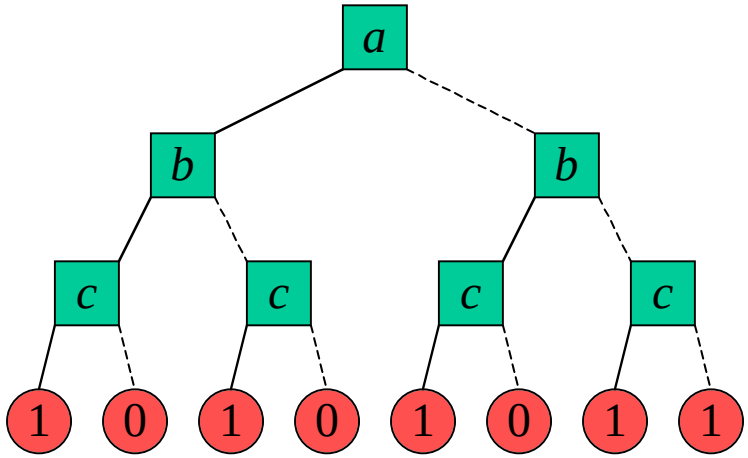
(*a* | *b*) => *c*

From Trees to Diagrams



- Fixed variable ordering

  "layered" tree



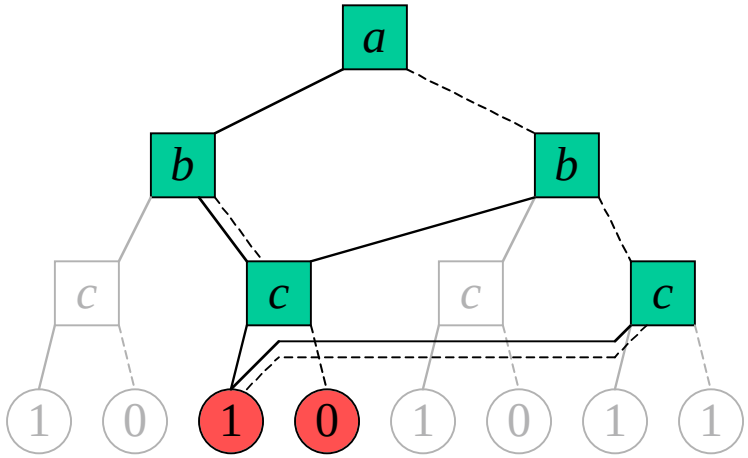
$(a \mid b) \Rightarrow c$

From Trees to Diagrams

- Fixed variable ordering

  "layered" tree

- Merge equal subtrees



$(a \mid b) \Rightarrow c$

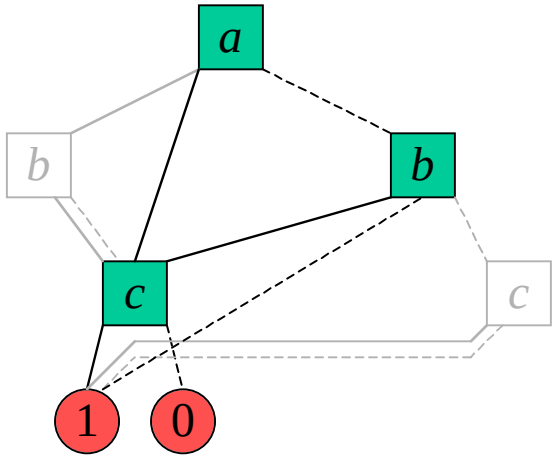
From Trees to Diagrams

- Fixed variable ordering

  "layered" tree

- Merge equal subtrees

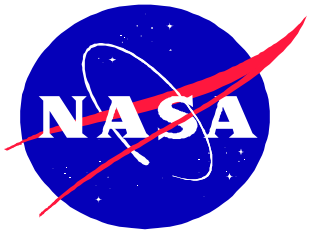
- Remove nodes with equal subtrees



$(a \mid b) \Rightarrow c$

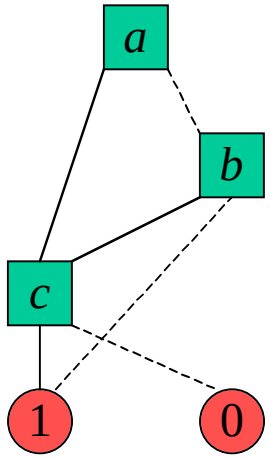
=> Ordered Binary Decision Diagram

[Ordered] Binary Decision Diagrams



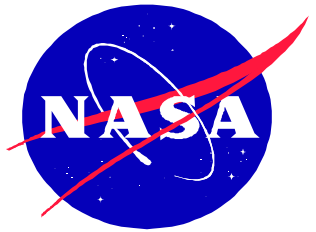
- [O]BDDS for short
- Unique normal form
  - for a given ordering and
  - up to isomorphism

  => compare in constant time (using hash table)



$(a \mid b) \Rightarrow c$

Computations with BDDs



- Negation $!f$:

  swap leaves 0 and 1.

- Boolean combinator $f\#g$:

  $(b\ ?\ f' : f'') \# (b\ ?\ g' : g'') = (b\ ?\ f'\#g' : f''\#g'')$

  cache results –> $O(|f|.|g|)$ time

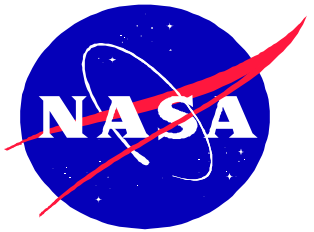
- Instantiation $f[b=1]$, $f[b=0]$:

  $(b\ ?\ f' : f'')[b=1] = f'$

- Quantifiers **exists** $b\ .\ f$, **forall** $b\ .\ f$:

  exists $b\ .\ f = f[b=1] \mid f[b=0]$

Variable Ordering



- Must be the same for all BDDs
- Size of BDDs depends critically on ordering
- Worst case: exponential w.r.t. #variables
  - sometimes exponential for any ordering

    e.g. middle output bit of n-bit multiplier
- Finding optimum is hard (NP-complete)

  => optimization uses heuristics

Transition Systems with BDDs

Given boolean state variables $v = b_1, ..., b_n$

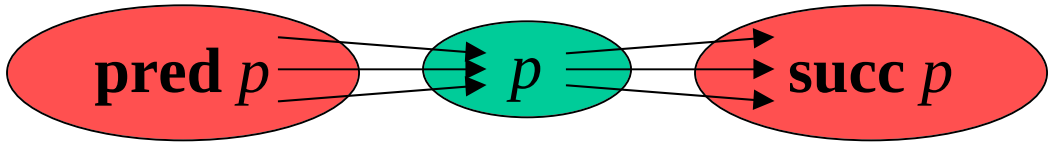
a set of states as a BDD $p(v)$

a transition relation as a BDD $T(v, v')$

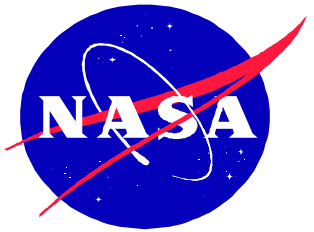
we can compute the predecessors and successors of p as BDDs:

(**pred** p)(v) = **exists** v' . $T(v, v')$ & $p(v')$

(**succ** p)(v) = **exists** v' . $p(v')$ & $T(v', v)$



Checking Formulas with BDDs



Functional evaluation as set of states:

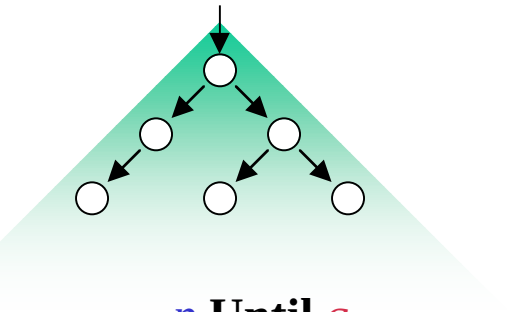
- for every formula p, build the BDD $p(v)$ of the set of states that satisfy p
- Top level: for a set of initial states I,

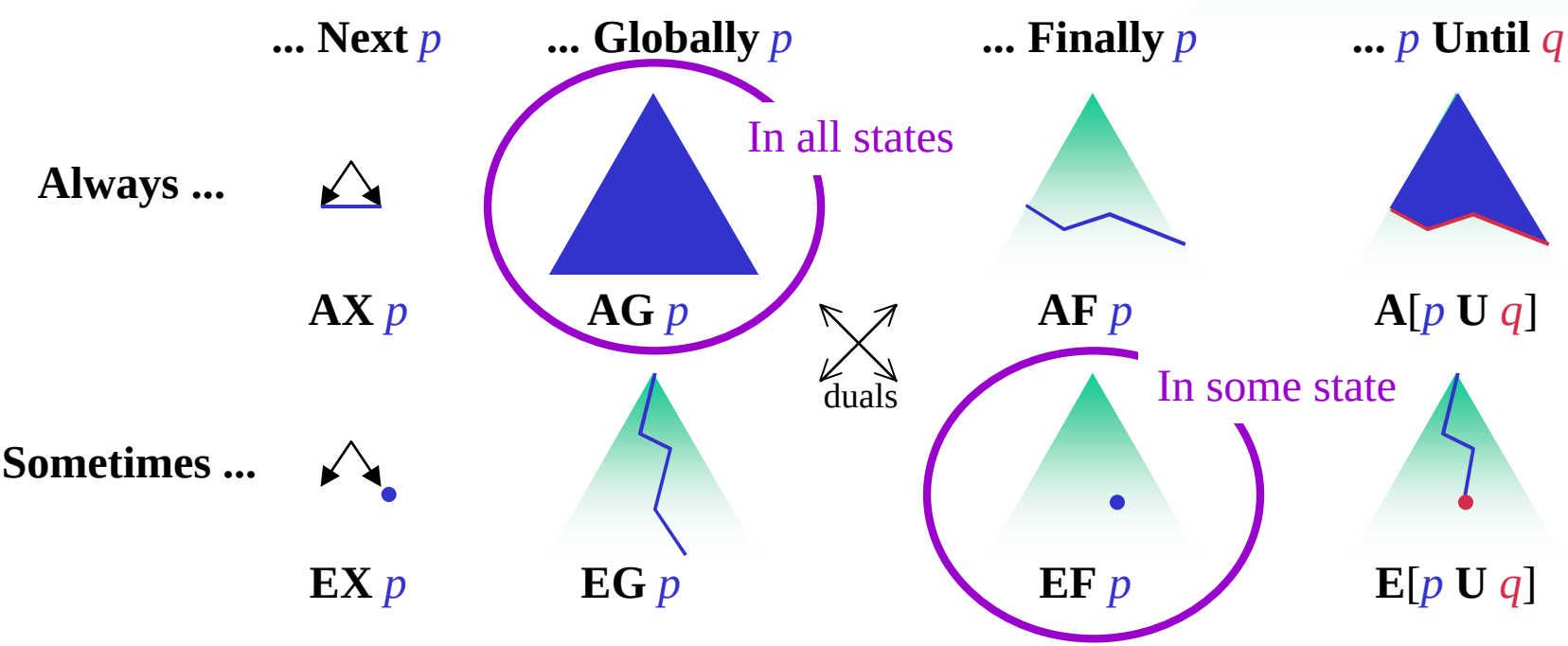
$$I \text{ satisfy } p \text{ iff } !p \,\&\, I = 0$$

- $p = \text{op}(q,r)$ => build $p(v)$ based on $q(v)$, $r(v)$

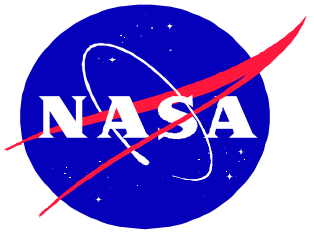
CTL temporal logic

Computation Tree Logic (branching):
Consider the tree of possible executions





CTL operators as BDDs



$(\mathbf{EX}\ p)(v) = (\mathbf{pred}\ p)(v) = \mathbf{exists}\ v'\ .\ T(v, v') \ \&\ p(v')$

$(\mathbf{EG}\ p)(v) = (\mathbf{gfp}\ U\ .\ p\ \&\ \mathbf{EX}\ U)(v)$

$(\mathbf{E}[p\ \mathbf{U}\ q])(v) = (\mathbf{lfp}\ U\ .\ q \mid (p\ \&\ \mathbf{EX}\ U))(v)$

All others can be expressed as **EX/EG/EU**

$\mathbf{EF}\ p = \mathbf{E}[1\ \mathbf{U}\ p]$

$\mathbf{AX}\ p = !\mathbf{EX}\ !p$

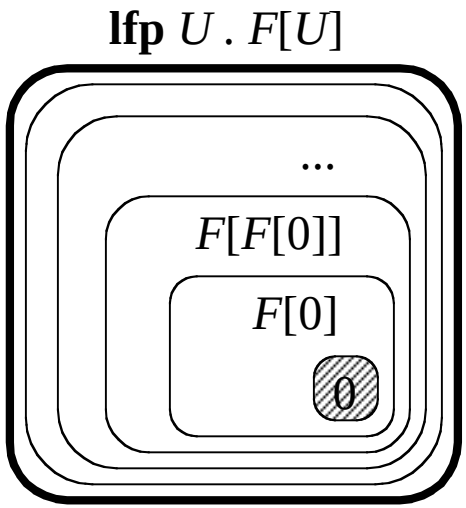
$\mathbf{AG}\ p = !\mathbf{EF}\ !p$

$\mathbf{AF}\ p = !\mathbf{EG}\ !p$

$\mathbf{A}[p\ \mathbf{U}\ q] = !\mathbf{E}[!q\ \mathbf{U}\ !p\ \&\ !q]\ \&\ !\mathbf{EG}\ !q$

Evaluating Fixpoints with BDDS

Compute **lfp** U . $F[U]$ as a BDD:



$U_0(v) = 0$

$U_1(v) = F[U_0](v) = F[0](v)$

...

$U_{n+1}(v) = F[U_n](v) = F^n[0](v)$

until $U_n(v) = U_{n+1}(v) = (\textbf{lfp}\ U\ .\ F[U])(v)$

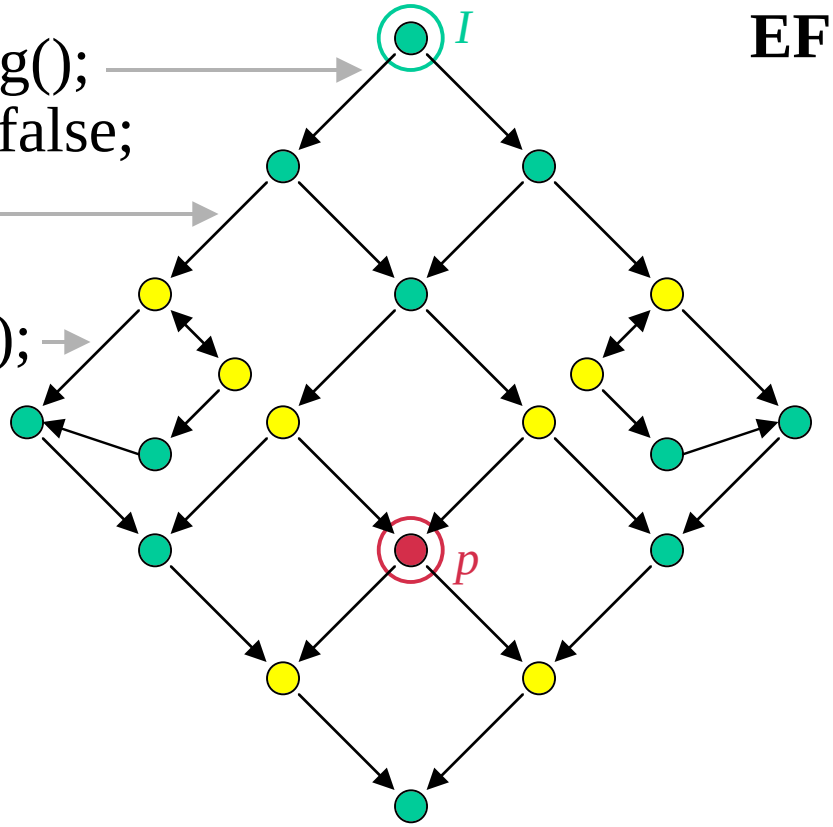
– Convergence assured because finite domain

– Dual construction for **gfp**

CTL with BDDS: Example

```
process P(id) {
  repeat {
    x=getFlag();
  } until x=false;
  setFlag();
  CS(id);
  resetFlag();
}

start P(1);
start P(2);
```



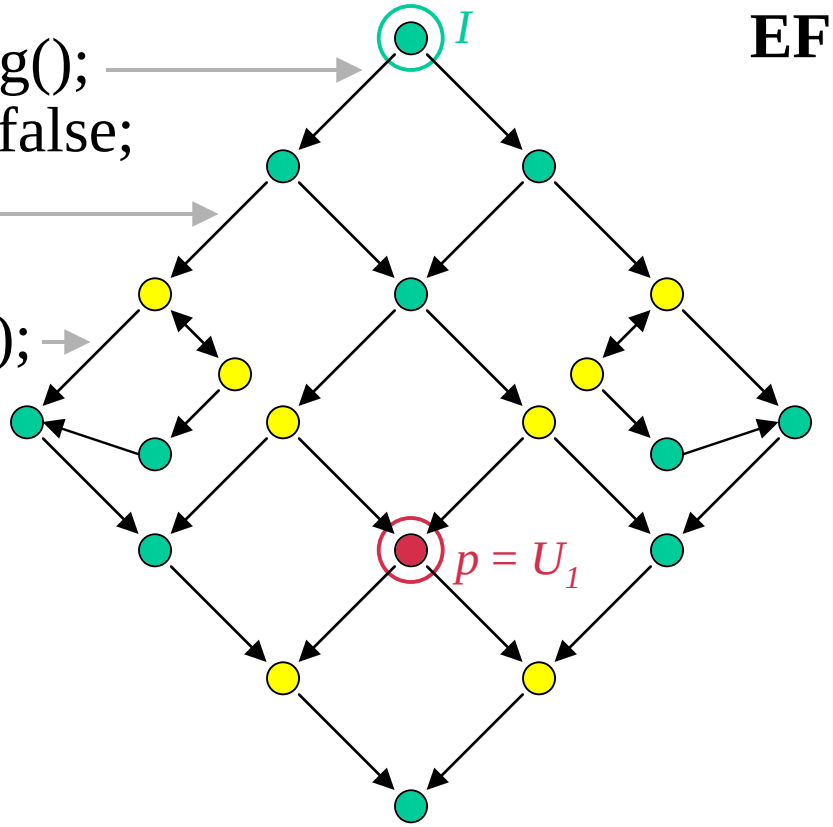
EF p = **lfp** U . p | **EX** U

$U_0 = 0$

CTL with BDDS: Example

```
process P(id) {
  repeat {
    x=getFlag();
  } until x=false;
  setFlag();
  CS(id);
  resetFlag();
}

start P(1);
start P(2);
```



EF p = **lfp** U . p | **EX** U

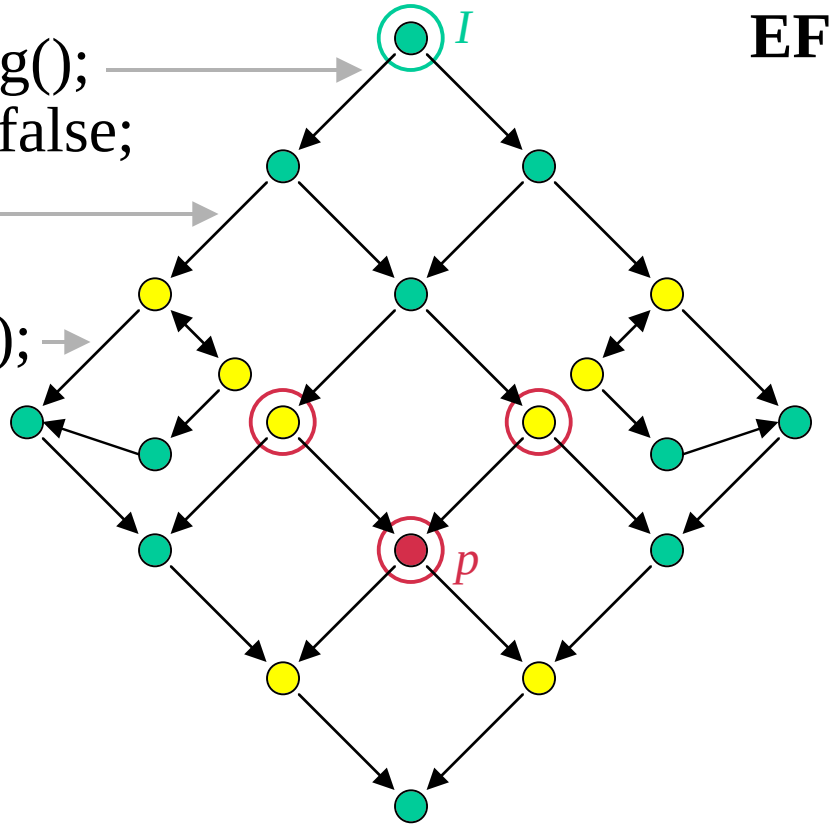
$U_0 = 0$

$U_1 = p$ | **EX** $U_0 = p$

CTL with BDDS: Example

```
process P(id) {
  repeat {
    x=getFlag();
  } until x=false;
  setFlag();
  CS(id);
  resetFlag();
}

start P(1);
start P(2);
```



EF p = **lfp** U . p | **EX** U

$U_0 = 0$

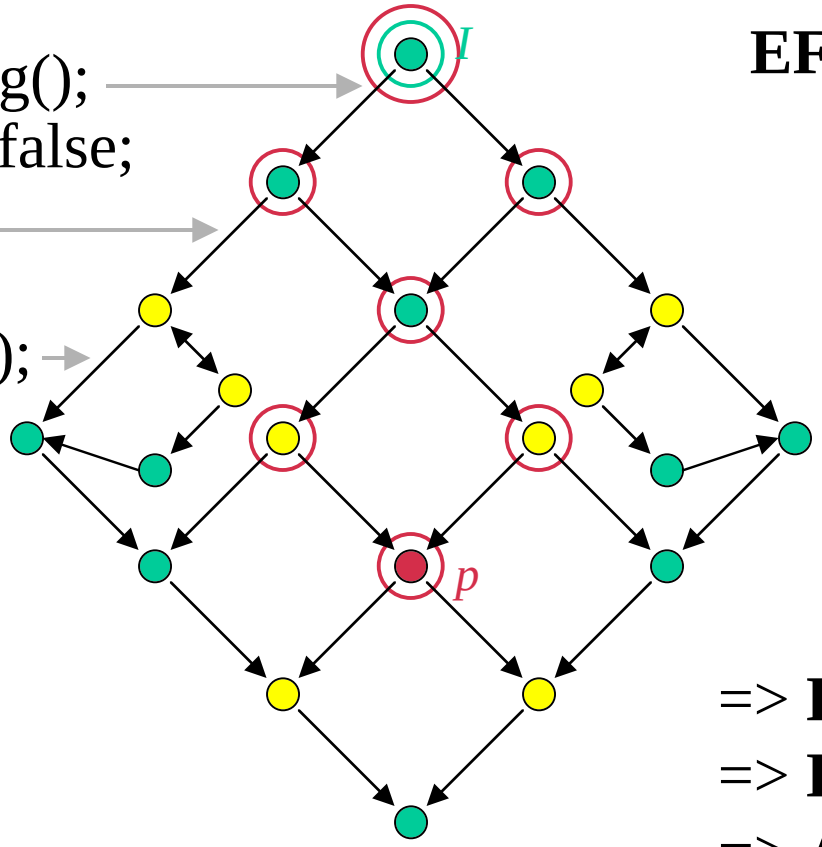
$U_1 = p$ | **EX** $U_0 = p$

$U_2 = p$ | **EX** U_1

CTL with BDDS: Example

```
process P(id) {
  repeat {
    x=getFlag();
  } until x=false;
  setFlag();
  CS(id);
  resetFlag();
}

start P(1);
start P(2);
```



EF p = **lfp** U . p | **EX** U

$U_0 = 0$

$U_1 = p$ | **EX** $U_0 = p$

$U_2 = p$ | **EX** U_1

...

$U_5 = p$ | **EX** U_4

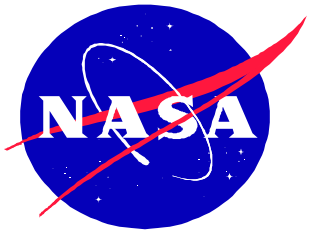
$U_6 = p$ | **EX** $U_5 = U_5$

=> **EF** $p = U_5$

=> **EF** p & $I \neq 0$

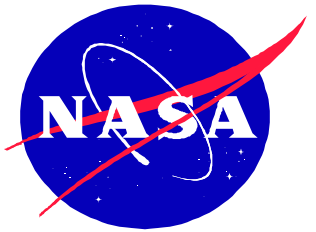
=> **AG** !p does not hold

Fairness, LTL



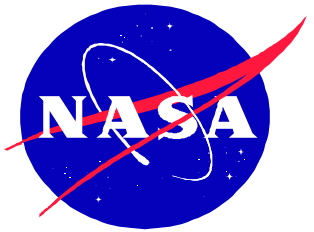
- CTL+fairness:
  - Only check executions where fairness conditions c_1, ..., c_n hold infinitely often
  - Symbolic evaluation: express c_1, ..., c_n as BDDs, modified algorithms for **EX**, **EG**, **EU**.
- Symbolic model checking of LTL
  - Convert LTL formula to Büchi automaton
  - Encode automaton in transition relation
  - Express acceptance condition in CTL+fairness

Bounded Model Checking



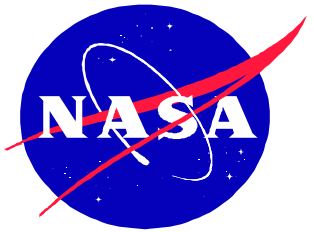
- Principle:
  - n+1 copies of state variables v_0, .., v_n
  - Unroll transition relation n times $T(v_{k-1}, v_k)$
  - Embed property to be satisfied
  - Verify satisfiability with SAT procedure
- Verifies traces up to length n
  - Iterate over values of n => breadth-first search
- No state space explosion (polynomial space)
- Usually fast (but worst case is exponential time)

Symbolic Model Checking Summary



- Principle: compute over sets of states encoded as BDDs.
- Can handle huge state spaces.
- CTL + fairness, LTL.
- Some tweaking may be needed.
  - variable ordering
- Some models blow up nevertheless.
- New alternative: SAT-based (bounded).

Symbolic Model Checking References



R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, 1986.

The seminal paper on Binary Decision Diagrams.

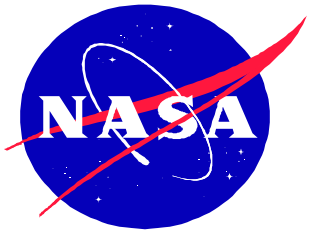
J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^20 states and beyond. *Information and Computation*, vol. 98, no. 2, 1992.

Survey paper on the principles of symbolic model checking.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *W. R. Cleaveland, ed., Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Amsterdam, March 1999.

Paper on SAT-based bounded model checking.

Symbolic Model Checking References (cont'd)

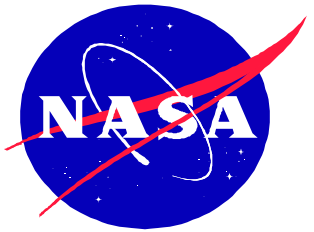


J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

Symbolic model checking of CTL with fairness.

E. Clarke, O. Grumberg, H. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, *Volume 10*, *Number 1*, February 1997.

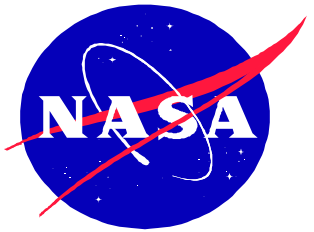
Verifying LTL using symbolic model checking.



Symbolic Model Checking Tools:

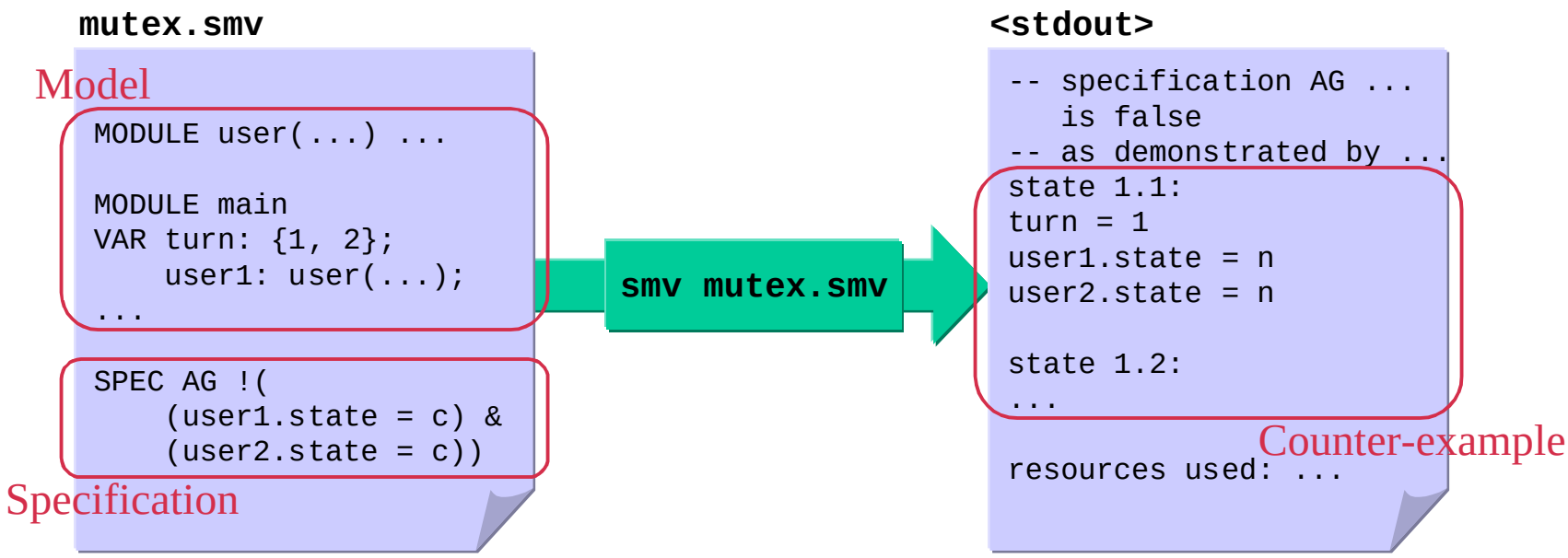
SMV

Overview



- **SMV** = **S**ymbolic **M**odel **V**erifier.
- Developed by Ken McMillan at Carnegie Mellon University.
- Modeling language for transition systems based on parallel assignments.
- Specifications in temporal logic CTL.
- BDD-based symbolic model checking: can handle very large state spaces.

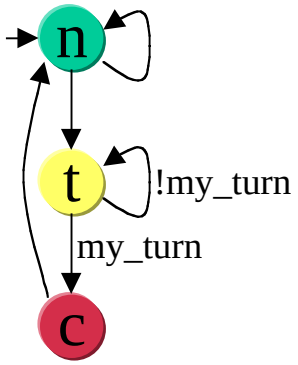
What SMV Does



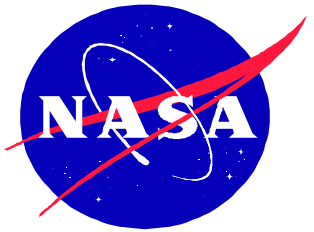
SMV Program Example (1/2)

```
MODULE user(turn,id,other)
VAR state: {n, t, c};
DEFINE my_turn :=
   (other=n) | ((other=t) & (turn=id));
ASSIGN
init(state) := n;
next(state) := case
   (state = n) : {n, t};
   (state = t) & my_turn: c;
   (state = c) : n;
   1 : state;
esac;

SPEC AG((state = t) -> AF (state = c))
```



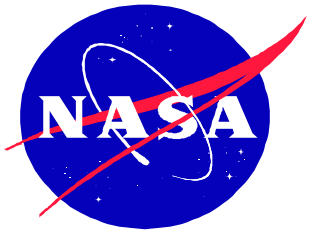
SMV Program Example (2/2)




```
MODULE main
VAR turn: {1, 2};
    user1: user(turn, 1, user2.state);
    user2: user(turn, 2, user1.state);
ASSIGN
init(turn) := 1;
next(turn) := case
   (user1.state=n) & (user2.state=t): 2;
   (user2.state=n) & (user1.state=t): 1;
   1: turn;
esac;

SPEC AG !((user1.state=c) & (user2.state=c))
SPEC AG !(user1.state=c)
```

Diagnostic Trace Example

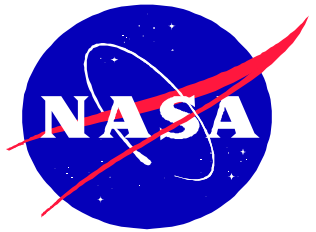



```
-- specification AG (state = t -> AF state = c) (in
   module user1) is true
-- specification AG (state = t -> AF state = c) (in
   module user2) is true
-- specification AG (!(user1.state = c & user2.state =
   c)... is true
-- specification AG (!user1.state = c) is false
-- as demonstrated by the following execution sequence
state 1.1:
turn = 1
user1.state = n
user2.state = n

state 1.2:
user1.state = t

state 1.3:
user1.state = c
```

The Essence of SMV

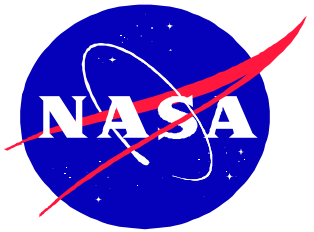


- The SMV program defines:
  - a finite transition model M (Kripke structure),
  - a set of possible initial states I (may be several),
  - specifications P_1 .. P_m (CTL formulas).
- For each specification P, SMV checks that

$$\forall\, s_o \in I \;.\; M, s_o \models P$$

Note: `SPEC !P` is not the negation of `SPEC P`:
both can be false (in some initial states),
both can be true (vacuously when $I=\varnothing$).

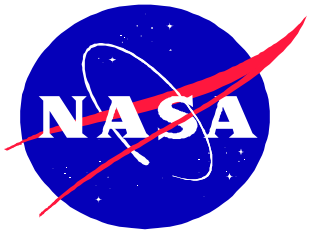
Variables and Transitions (Assignment Style)




```
VAR state: {n, t, c};
ASSIGN
init(state) := n;
next(state) := case
   (state = n) : {n, t}; ...
esac;
```

- Finite data types (incl. numbers and arrays).
- Usual operations `x&y`, `x+y`, etc., `case` statement.
- All assignments are evaluated in parallel.
- No control flow (must be simulated with vars).
- SMV detects circular and duplicate assignments.

Defined Symbols

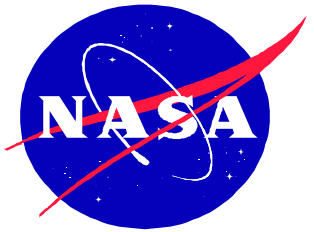



```
DEFINE my_turn :=
   other=n | (other=t & turn=id);
ASSIGN
next(state) := case ...
   (state = t) & my_turn:  c; ...
esac;
```

- Defines an abbreviation (macro definition).
- No new state variable is created

  => no added complexity for model checking.
- No type declaration is needed.

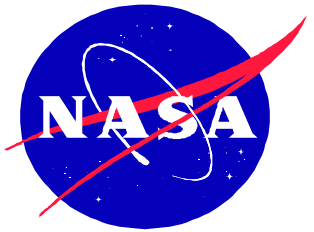
Modules




```
MODULE user(turn,id,other)
VAR ...
ASSIGN ...
MODULE main
VAR user1: user(turn,1,user2.state);
    ...
```

- Parameters passed by reference.
- Top-level module `main`.
- Composition is synchronous by default: all modules move at each step.

Records

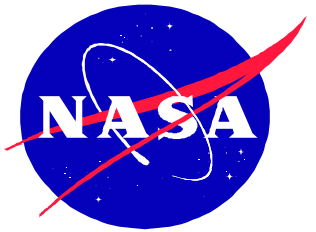


Modules without parameters and assignments.

```
MODULE point
VAR  x : {0,1,2,3,4,5};
     y : {0,1,2,3,4,5};

MODULE main
VAR  p : point;
ASSIGN
   init(p.x) := 0; init(p.y) := 0;
   ...
```

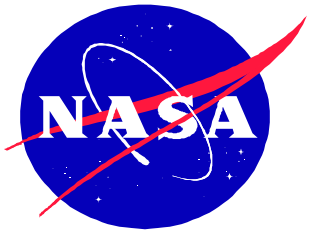
Processes




```
VAR node1: process node(1);
    node2: process node(2);
```

- Composition of processes is asynchronous: one process moves at each step.
- Boolean variable `running` in each process
  - `running=1` when that process is selected to run.
  - Used for fairness constraints (see later).

Specifications




```
SPEC AG ((state = t) -> AF (state = c))
```

"Whenever state `t` is reached, state `c` will always eventually be reached."

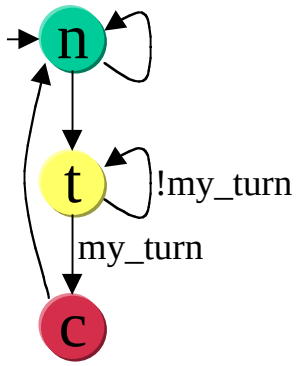
- Standard CTL syntax:

  `AX p`, `AF p`, `AG p`, `A[p U q]`, `EX p`, ...

- Can be added in any module.
- Each specification is verified separately.

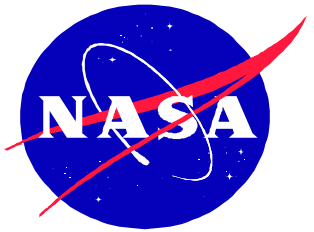
Fairness

```
MODULE user(turn,id,other)
VAR ...
ASSIGN ...
SPEC AG AF (state = c)
FAIRNESS (state = t)
```



- Check specifications, assuming fairness conditions hold repeatedly (infinitely often).
- Useful for liveness properties.
- Fair scheduling: `FAIRNESS running`

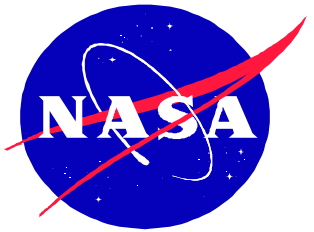
Variables and Transitions (Constraint Style)




```
VAR pos: {0,1,2,3,4,5};
INIT pos < 2
TRANS (next(pos)-pos) in {+2,-1}
INVAR !(pos=3)
```

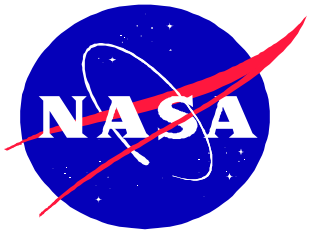
- Any propositional formula is allowed
  => flexible for translation from other languages.
- `INVAR p` is equivalent to `INIT p` `TRANS next(p)`
  but implemented more efficiently.
- Risk of inconsistent models (`TRANS p & !p`).

Well-Formed Programs?



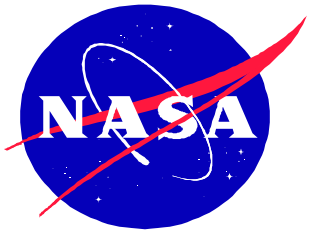
- In assignment style, by construction:
  - always at least one initial state,
  - all states have at least one next state,
  - non-determinism is apparent (unassigned variables,set assignments, interleaving).
- In constraint style:
  - `INIT` and `TRANS` constraints can be inconsistent,
  - the level of non-determinism is emergent from the conjunction of all constraints.

Inconsistency



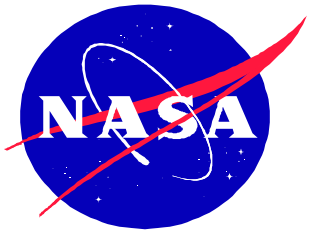
- Inconsistent `INIT` constraints
  => inconsistent model: no initial state.
  - `SPEC 0` (or any `SPEC P`) is vacuously true.
- Inconsistent `TRANS` constraints
  => deadlock state: state with no next state
  => transition relation is not complete.
  - SMV does not work correctly in this case.
  - SMV will detect and report it.

Variable Ordering



- BDDs require a fixed variable ordering .
  - Critical for performance (BDD size).
  - Best one is hard to find (NP-complete).
- SMV does not optimize by default but
  - can read, write ordering in a file,
  - can search for better ordering on demand.

Re-ordering Variables



Using command line options:

`smv -o demo.var`

Outputs variable ordering to `demo.var`.
`demo.var` is text, can be re-ordered manually.

`smv -i demo.var`

Inputs variable ordering from `demo.var`.

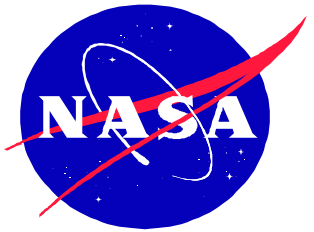
`smv -reorder`

Does variable re-ordering when BDD size exceeds a certain (configurable) limit.

`smv -reorder -oo demo.var`

Outputs to `demo.var` after each change.

Re-ordering Variables Method for Tough Cases



Problem (Livingstone ISPP model):

```
smv ispp.smv
```

-> Memory overflow.

```
smv -reorder ispp.smv
```

-> keeps re-ordering again and again...

Solution:

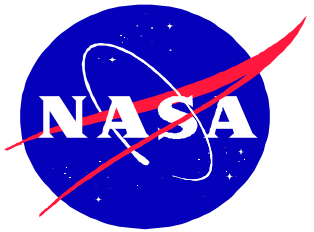
```
smv -reorder -oo ispp.var ispp.smv
```

Wait until "enough" re-ordering (statistics).

```
^C
smv -i ispp.var ispp.smv
```

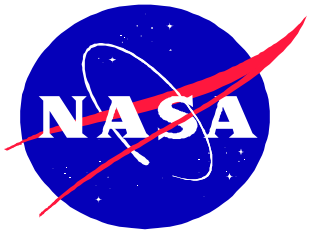
-> Goes to completion (10^{50} states).

Availability



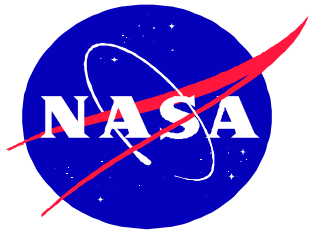
- Freely downloadable.
- Source or binaries for Unix (SunOS4, SunOS5, Linux x86, Ultrix).
- Windows NT port (Dong Wang).
- see http://www.cs.cmu.edu/~modelcheck/smv.html

NuSMV



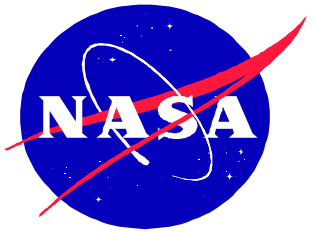
- From ITC-IRST (Trento, Italy) and CMU.
- New version of SMV, completely rewritten:
  - Same language as SMV.
  - Modular, documented APIs, easily customized.
  - Specifications in CTL or LTL.
  - Graphical User Interface.
  - Usually faster but uses more memory.
- See http://sra.itc.it/tools/nusmv/index.html

Other Related Tools



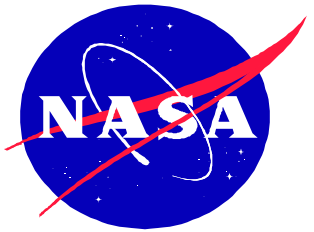
- **Cadence SMV** (Cadence Berkeley Labs)
  - From Ken McMillan, original author of SMV.
  - Supports refinement, compositional verification.
  - New language but accepts CMU SMV.
  - see http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/
- **BMC** = Bounded Model Checker (CMU)
  - Uses SAT procedures instead of BDDs: bounded depth but usually faster, less memory.
  - Simple SMV-like language (no modules).
  - Early beta version.
  - see http://www.cs.cmu.edu/~modelcheck/bmc.html

SMV Summary



- BDD-based symbolic model checker.
- Modeling language based on synchronous transition systems.
- Constraint style: more versatile, less strict => good for use as back-end tool.
- 1st generation: CMU
- 2nd generation: Cadence, NuSMV
- Variant: BMC (SAT based)

SMV References



Ken McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.

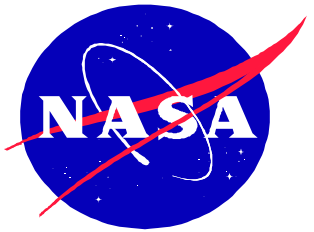
Based on Ken McMillan's PhD thesis on SMV.

Ken L. McMillan. The SMV System (draft). February 1992.
http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps

The (old) user manual provided with the SMV program.

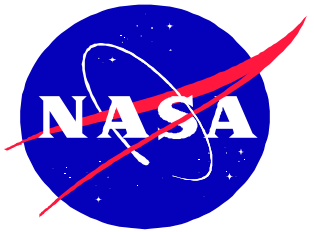
A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *N. Halbwachs and D. Peled, eds., Proceedings of International Conference on Computer-Aided Verification (CAV'99)*, LNCS 1633:495-499, Springer Verlag.

Survey paper on NuSMV.



Symbolic Model Checking Applications in Software

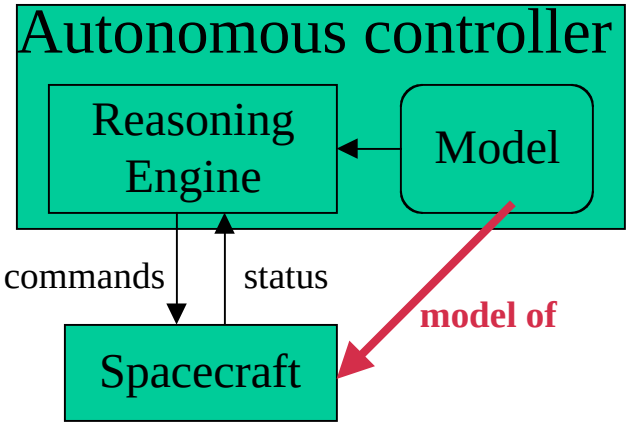
Applications of Symbolic Model Checking



- Used in industry for hardware design
  - Commercial tools (Cadence)
  - Fits well with boolean modeling
- Some success stories in protocol design
  - Cache coherence of IEEE Futurebus+
  - HDLC
- Research stage for software design
  - Gap between programming/design language and verification modeling language.

Model-Based Autonomy

- Unattended control of a complex device (e.g. a spacecraft)
- Based on AI technology
- General reasoning engine + application-specific model
- Use model to respond to unanticipated situations

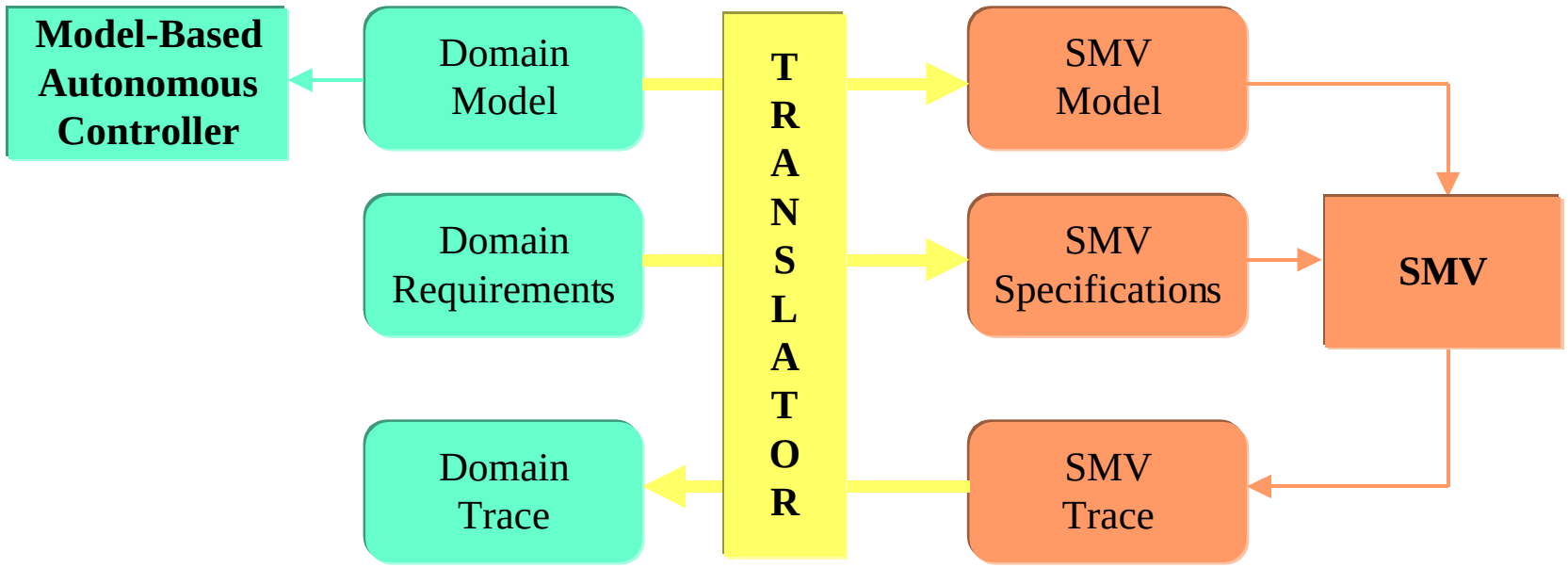


=> Verify the model !

Verification of Autonomy Models

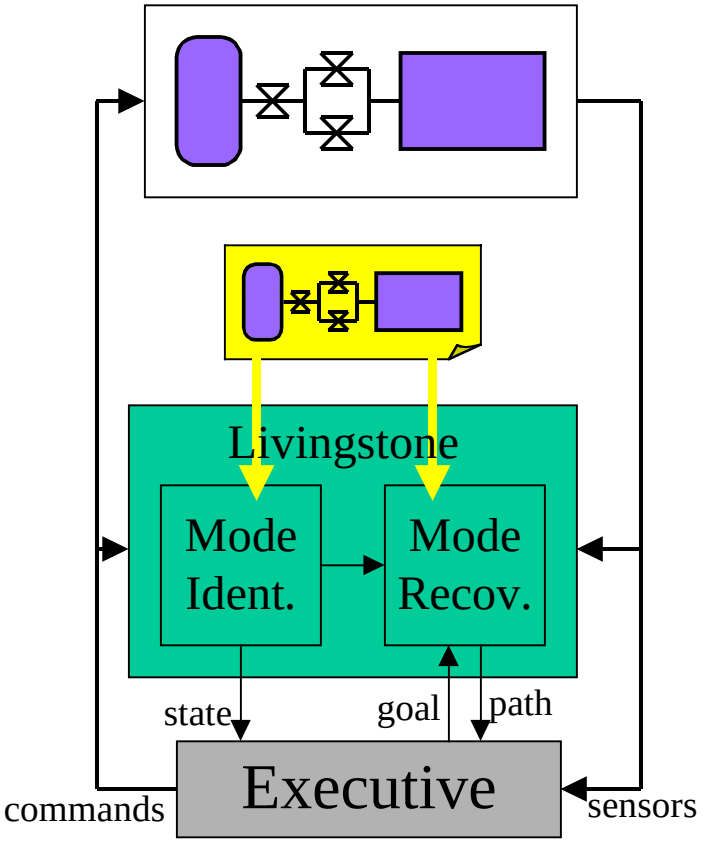
Model-Based Autonomy

Verification



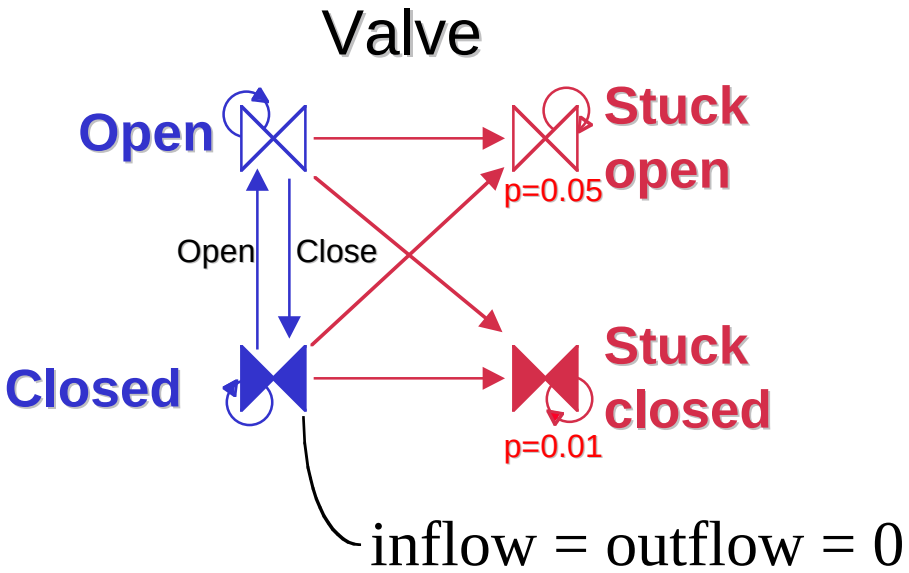
The Livingstone Diagnostic System

- Mode identification & recovery:
  - identify current state (including faults)
  - find path to goal state
- Model-based
- From NASA Ames
- Run in space (DS- 1, May 1999)



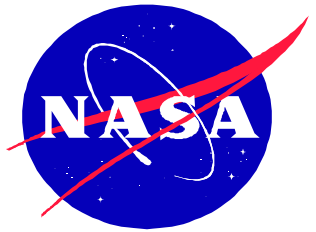
Livingstone Models

- Models = concurrent transition systems
- Qualitative values => finite state
- Nominal/fault modes
- Probabilities on faults



Courtesy Autonomous Systems Group, NASA Ames

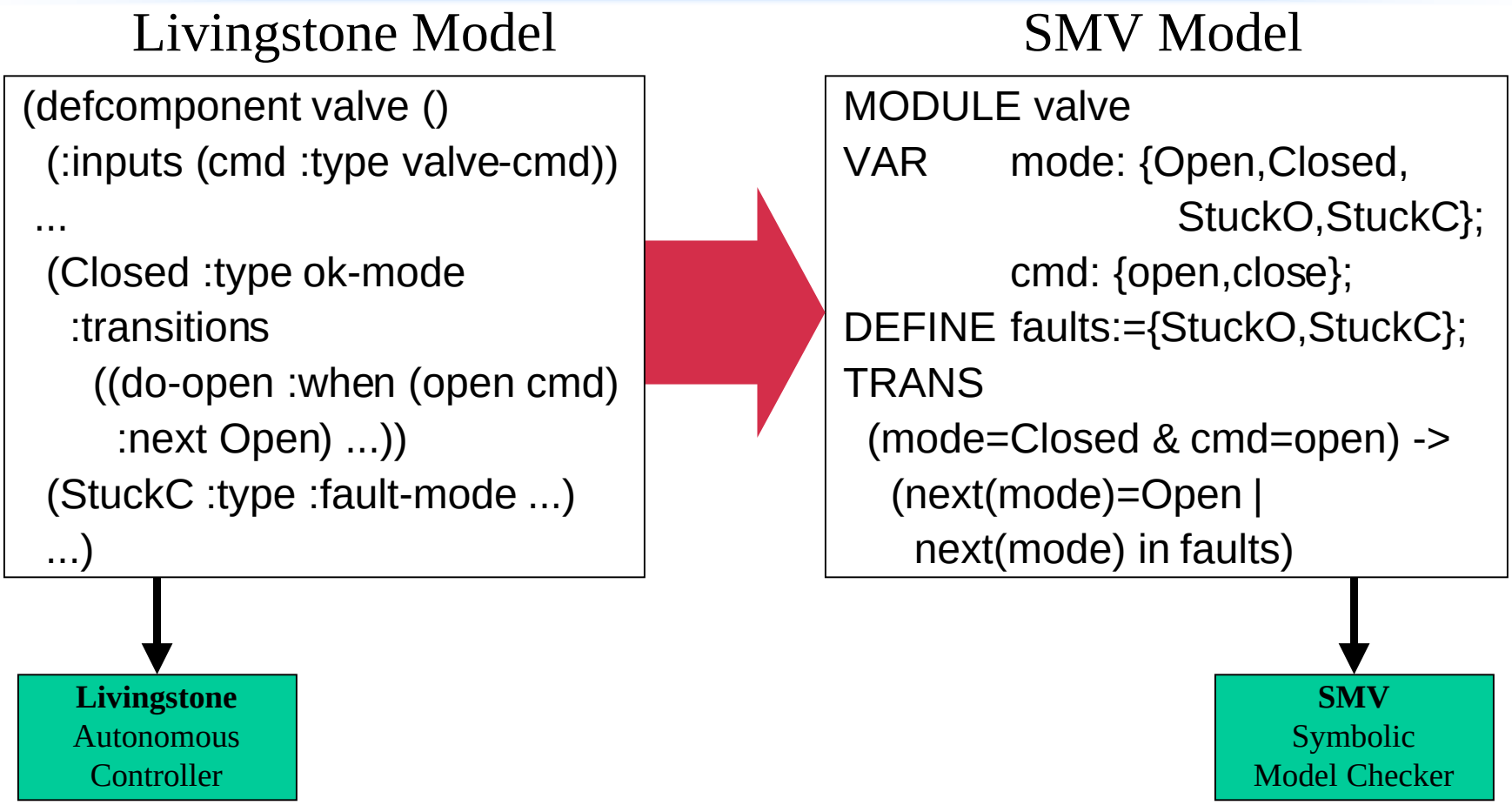
Formalizing the Model



Livingstone model	SMV model
atomic component	module
compound module	module
variables	scalar variables
structures	module variables
mode transitions	TRANS
model constraints	INVAR
initial state	INIT

Main difficulty is translating Livingstone's flat name space

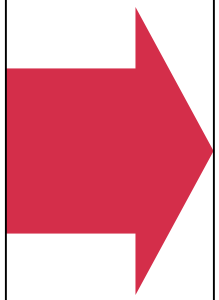
Translating Models



Translating Requirements

Livingstone Requirement

```
(defverify ...
  (:specification
    (always (globally (implies
      (not (broken))
      (exists (eventually
        (high flow-in))))))
```



SMV Requirement

```
SPEC AG (
  (!broken) ->
  EF (ISPP.valve.flow-in = high))
```

- Declaration (defverify ...) added to the Livingstone model.
- Temporal logic formulas (CTL) in Livingstone syntax.
- Auxiliary predicates (e.g. failed component).
- High-level property patterns (e.g. reachability of modes).

Application
In-Situ Propellant Production

- Use atmosphere from Mars to make fuel for return flight.
- Livingstone controller developed at NASA Kennedy.
- Components are tanks, reactors, valves, sensors...
- Exposed improper flow modeling.
- Very "loose" state space:
  - 10^{50} states
  - all states reachable in 3 steps

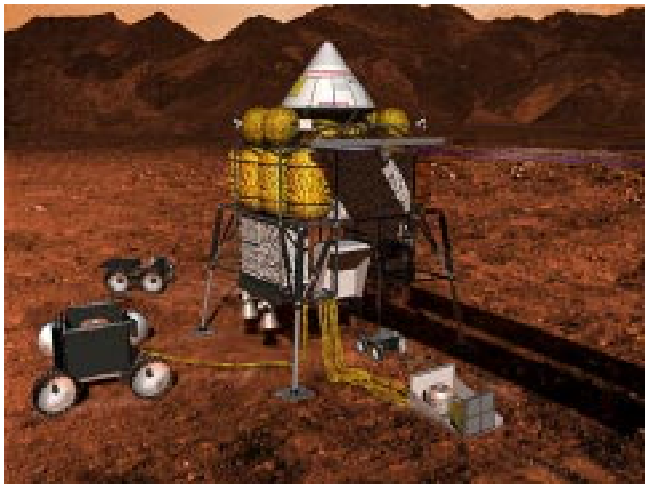
Mars atmosphere

on-board

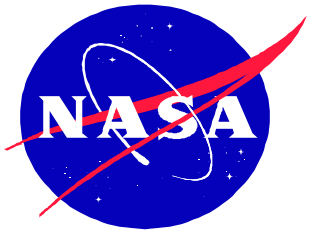
$$CO_2 + 2H_2 \longrightarrow CH_4 + O_2$$

fuel

oxidizer

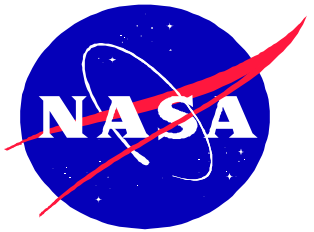


TDL:
Task Description Language



- Extension of C++
- Task decomposition, task synchronization, monitoring, exception handling
- From Carnegie Mellon University
- Used for robot control architectures

Formalizing TDL

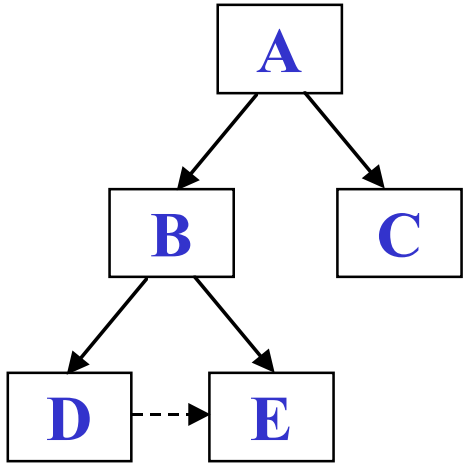


TDL model	SMV model
task	module
task/subtask relationship	module variables
task state	scalar variables
state transitions	ASSIGN
temporal constraints	INVAR and parameters
asynchronous nature	***PROCESS variables and FAIRNESS constraints***

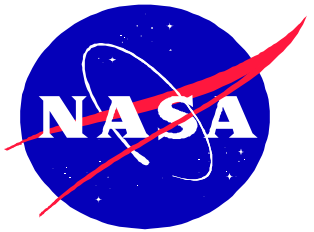
Verifying Task Descriptions

Work by Reid Simmons (CMU)

- Can verify temporal properties of hierarchical tasks
  - deadlock, safety, liveness, …
  - can handle conditional execution
- In progress:
  - monitoring and exception handling
  - iteration and recursion

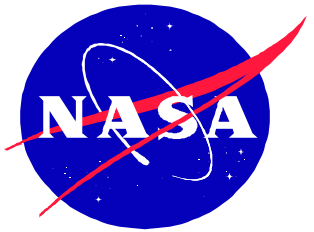


Applications of SMV Summary



- Symbolic model checking: OK for hardware, quid for software?
- Needs translation from programming language to verification language and back!
- 2 examples for autonomy software using SMV.

Applications of SMV References



C. Pecheur and R. Simmons. “From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts”. *First Goddard Workshop on Formal Approaches to Agent-Based Systems*, NASA Goddard, April 5-7, 2000.

Verification of Livingstone with SMV.

R. Simmons and C. Pecheur. “Automating Model Checking for Autonomous Systems”. *AAAI Spring Symposium on Real-Time Autonomous Systems*, Stanford CA, March 2000.

Verification of TDL with SMV.